X68040 turbo

A Story of Making "After X68030"

著・BEEPs

SOFT BANK

著●BEEPs

SOFT BANK

●お問い合わせ
本書の内容に関する質問は、電話ではなく、下記宛に郵送でお願いします。
〒103　東京都中央区日本橋浜町 3-42-3
ソフトバンク株式会社　出版事業部
ハードウェア活用書編集部

■はじめに

「X68/040turbo」この名前を見て、オッと思った人もいるでしょう。しかし、残念ながら、X68シリーズの新機種ではありません。これは、私の愛機のことを勝手にそう呼んでいるだけです。でも、この名前はダテじゃありません。[*1]

本体はX68030ですが、マザーボードのプロセッサソケットには、68EC030のかわりに040tubro[*2]と名付けたドータボードが挿さっており、この上に、名前のとおり68040が搭載[*3]されているのです。

もちろん、040turboは既製品ではありません。個人的な68040への思い入れで作った手作りハードウェアです。しかし、パソコン通信を通して多くの人の賛同を得たことから、プリント基板を起こして一般公開し、参加者の数も50人を超えるまでになっています。

そして、公開されたフリーソフトウェアが多くの人の手でさらによいものに磨き上げられていくように、この040turboも参加者みんなの手によって磨かれてきました。今やHuman68k[*4]の使用はほとんど支障がないレベルにまでなっていますし、NetBSD[*5]も68030バージョンの移植と並行して、68040バージョンの移植もいち早く行われました。

しかし、ここまでくるのは決して簡単ではありませんでした。この本は、X68030登場からほぼ1年、040turboに明け暮れた日々を綴った、040turboのメイキングストーリーです。

「040turbo製作編」は、040turbo誕生のストーリーです。
「番外奮闘編」は、040turbo対応のプログラムを作ってくれた方に寄稿してもらった、それぞれの奮闘記です。

付録1「040turbo取扱説明書」は、040turboの「取扱説明書」を再編集したものです。040turboに関するすべての技術情報が、ここに詰まっています。

1994年4月

BEEPs

＊1
察しはつくと思いますが、この名前、8ビットマシンの名機、「X1turbo」にちなんだものです。

＊2
最初は「TURBO040」という名前でしたが、某マシンのアクセラレータにこういう名前の製品があったので、ひっくり返して、こう命名しました。

＊3
68030も搭載できるようになっており、スイッチで切り替えて使うことができます。

＊4
ver3.01、ver3.02にパッチを施しています。

＊5
フリーのUNIXの1つ。これは、ソースのある強みです。

■CONTENTS

番外奮闘編 245

付録 309

Cover Photo…Fumio SAITOH
Cover Design…Masaki KATSUMATA
Illustration…Shigeru ISHIGURO

040turbo製作編

第1章

X68030がやってきた

X68000からX68030へ

X68030がやってきたときのことは今でもよく覚えています。

X68000XVIへの買い換えを見送って、ひたすら32ビットのX68000を待ち続けた私には、X68030の噂を聞いて〝68030〟という点に割りきれない思いを持ちつつも[*1]、とてもそれを見送る余裕はありませんでした。正式発表も待たずに、詳細未定、値段未定の段階で、秋葉原のT店に予約[*2]を入れていたのです。

それからは、噂に一喜一憂しつつ、正式発表の日を指折り数えて待ちました。

そして、とうとう登場してきたそのマシンは、数ある噂のなかでも最も下の[*3]スペックと、最も上の値段でした。もともと、PC/AT互換機やMacintoshに浮気しそうになる心を押さえるために、予約を入れて待ち続けたようなものですから、実際に発売が開始されたと聞いては、いてもたってもいられません。

予約したT店に電話を入れると、入荷状況がよくなくて、いつ渡せるかわからないとの返事。なんのための予約だと思いながらも、最も信頼できそうなT店のこと、ここにないのであれば、どこにもないんだろうなと自分に言い聞かせました。

しかし、その夜、パソコン通信をのぞくと、すでに購入したという書き込みを発見。それも、秋葉原で売っていたという話です。

なんだなんだ、予約が後回しにされているのか？

次の日、いきり立ってまた電話すると、昨日の電話の後、何台か入荷したので、すぐ発送したとの返事。それまでの怒りもどこへやら、電話しながら顔がニヤケてくるのがわかります。

そうと聞いたら、すぐさま行動開始です。X68030のハードディスクはハーフピッチタイプですが、今使っているX68000のハードディスクはすべてフルピッチタイプなので変換ケーブルが必要です。急いで買いに行こうと部屋を出たとき、飛脚のマークのトラックが到着したのが見えました。そして、4階のわが家までおじさんが肩にかついで運んできたのは、横に「32」と大きく描かれた茶色のダンボール箱でした。

＊1
すでに68030は古いという思いがありましたので。

＊2
ちなみに、私は名古屋に住んでいますが、早く手に入るかもしれないと、わざわざ秋葉原の店に予約を入れました。その店の予約第2号でした。私と同じような思いの人がほかにも1人はいたというわけです。

＊3
激烈な低価格競争を繰り広げているPC/AT互換機やMacintoshだったら、ディスプレイとハードディスク込みのフルセットが買えてしまうくらいの値段でフロッピーモデルの本体しか買えないのですから、予約していなかったら、私もほかのマシンに転んでいたかもしれません。

部屋を出ようとしたら、横に大きく「32」と書かれた茶色の箱をかついだおじさんがやってきた。

最初にやったこと

いちおう最初は、X68030がちゃんと起動するかどうかを調べる意味もあって、同封のフロッピーディスクを使って動かしてみましたが、動作確認もそこそこにX68030の右側タワーの分解を始めました。X68030はマザーボードがシールドにガッチリ覆われているので、分解しないとマザーボードをほとんど拝むことができません。

X68000のときは内蔵のメモリボードを取り付けるときに開けたくらいで、その後、クロックアップの話題が盛り上がるまで、およそ4年くらい、一度も分解したことがありませんでした。X68000の盛りだくさんの機能を使うのが楽しくて、ソフトウェアだけで満足していたからでしょう。

これに対し、X68030は機能的にX68000とまったく変わらなかったこともあって、もっぱら興味は32ビット化されたプロセッサまわりのハードウェア一点にありました。そういう事情もあって、一刻も早く自分の目で68030の載っているマザーボードを見てみたかったのです。写真1.1が、X68030を開けてみたところです。

X68030のマザーボードの第一印象は、部品が多く、複雑だなという感じでした。同じモトローラのプロセッサを使っていても、Macintoshのマザーボードは、コンセプトの違いもあるのでしょうが、非常にシンプルです。特に、同時期に発表され、同じ25MHzの68030を搭載したMacintosh LC 3とX68030を比べると、この違いは顕著です。

こんな複雑な作りをしていたら、X68030がバカ高い値段になってもしかたがないのかなと妙に納得したりしてしまいました。

といっても、所狭しと部品が並んでいるX68030のマザーボードもマニアックな感じで好きです。昔、MZ-80Kのマザーボードを眺めていた[*1]ので、その感覚が染み付いているのかもしれませんが、X68030のマザーボードを眺めて、

なかなかカッコイイじゃない。

と、しばし悦に入っていました。

余談ですが、初代のX68000のマザーボードをはじめて見たときは、壮絶な作りにびっくりしました。普通コンピュータ基板といったら整然と部品が並んでいて、その間を改造のための細い線が走っている程度なのですが、私のX

＊1
MZ-80Kは、車のボンネットを開けるように、本体をパカッと開けることができました。この中には、8ビットマイクロプロセッサの傑作、Z80を戴くマザーボードが鎮座しており、意味もなく開けては、眺めていたものです。

68000の中はテレビの配線かと思われるような太い線が這い回り、ICの足にコンデンサが束になって直付けされていたのです。もちろん、私がやったわけではありません。もともと、そういうマザーボードだったのです。

そこへいくと、X68030のマザーボードは改造線など1本もなく、非常に美しい基板でした。まあ、発表がかなり遅れましたから、出荷までにマザーボードをきれいに改版したのでしょう。

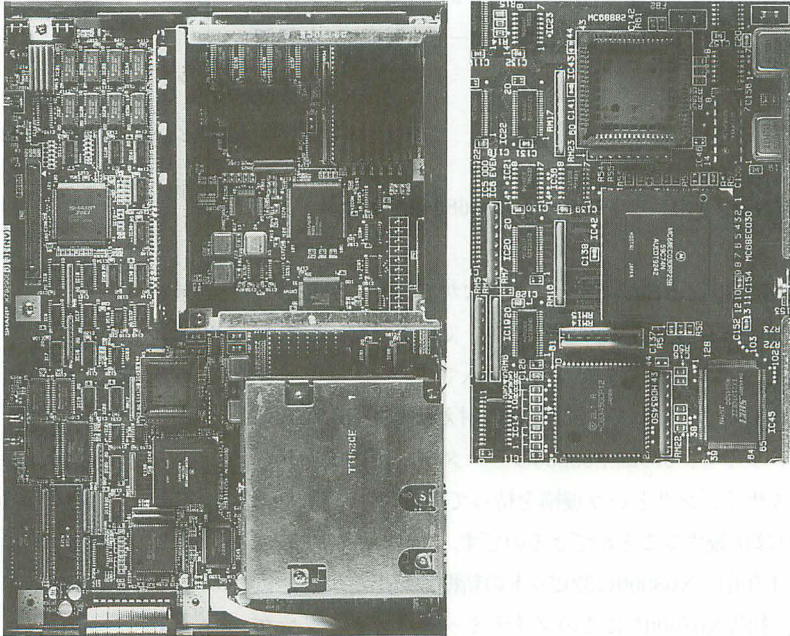
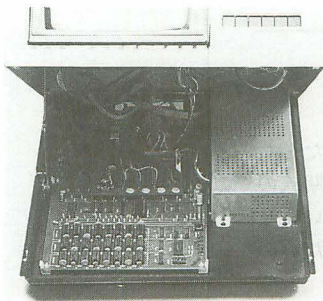



写真1.1　上：X68030のマザーボード　下：MZ-80Kのフタを開けたところ　右：68030まわり

X68030はいいマシンか

5年以上も待たせたのに、グラフィックやサウンド機能などに関してはX68000とまったく変わらず、売りであるプロセッサの32ビット化にともなうパワーアップといってもX68000比にすぎません。ちまたにある他のマシンはさらに上を行っていますから、残念ながら、X68000ユーザーでもX68030を評価しない人は多いようです。*1

かくいう私も、

インテル系は486が主流なんだもの、モトローラ系なら68040だよな。

と思っていましたから、噂*2の信憑性が高まっていくなかで、もしかしたらX68040になるんじゃないか、というかすかな望みを抱いていました。ですから、X68030の正式発表を見て、ガッカリしたうちの1人です。

しかし、X68030は堅実に32ビット化して、25MHzに耐えうる性能になったことは評価できます。というのも、68020や68030はデータバスが16ビットでも使えるダイナミックバスサイジングという機構を持っているので、X68000にちょっと回路を追加すれば接続することができるのです。

実はX68030が登場する1年前、X68000に32ビットの新機種が出なかったことに失望して、前の愛機、初代X68000*3にこのダイナミックバスサイジング機構を使って68020を載せたことがあります。

＊1
X68030の発表を機にX68000から他機種に移ってしまった人は多いようです。

＊2
噂では、たいてい、とんでもないスペックのマシンになっていたりするのですが、今回の噂はプロセッサが68030という点だけは共通していました。

＊3
X68000XVIもいいマシンでしたが、私の望んでいたものではなかったので買い換えには至りませんでした。

C O L U M N

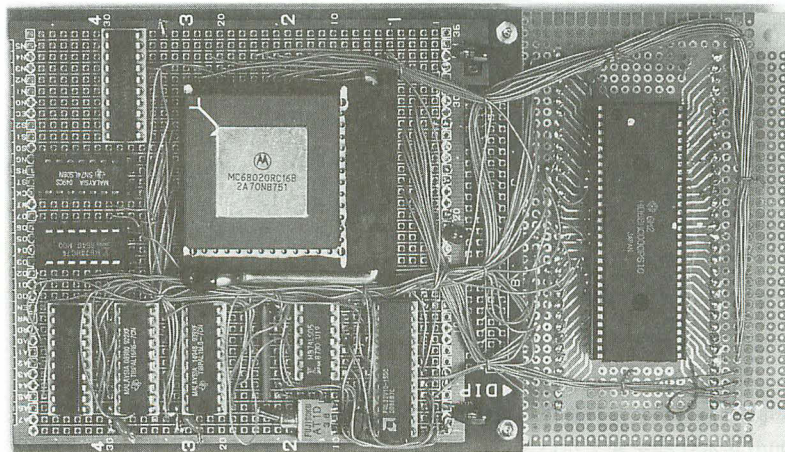
幻のX68020

噂では、何年も前からX68020があったけど、お蔵入りになっていたとか。調べてみると、確かにX68030のハードウェアは、バースト転送や同期式のバスアクセスなど、68030で強化されたハードウェア機能は使われていません。X68020をベースにして、ちょこっと68030用の回路がつけてあるだけのようです。

しかし、もしこの噂が本当なら、X68020でいいから、もっと早く発表してほしかったものです。

ハードウェアの製作は比較的簡単でした。『68020ファミリ活用の実際』（電波新聞社）という本に68020を68000につなぐための回路が載っていたので、これをX68000用にモディファイし、68000と68020を切り替えて使えるようにしたのです。後は力業で、写真1.2のようなものを作り上げました。むしろHuman68k ver2.03を解析して、68000と68020の非互換部分をパッチ*[1]するソフトウェアを作成する作業のほうが大変だったくらいです。

苦労の末、なんとかX68000の10MHzの16ビットバスの上で動いた68020でしたが、結果はほとんど速くなりませんでした。実は、前にもパソコン通信で68020をX68000に載せた話を聞いたことがありました。その結果もほとんど速くはならなかったそうですが、そう聞いてはいても、自分の手でやってみたかったのです。68020側のクロックアップや、32ビットのローカルメモリなどを持たせれば性能が上がるかもしれませんが、とても、そこまでやる気力はありませんでした。

結局、この試みは「68020が動いた」*[2]という自己満足だけで終わり、ちゃんとした32ビットデータバスのX68000の後継機種が出るのを待つことになったのです。

＊1
後にHuman68k ver3をこのマシンで使ったら、ノーパッチで動きました。

＊2
しばらく使っていると暴走してしまいましたが、その原因は追究しませんでした。ちなみに、スペースハリアーはこのマシン上でずっと動いていました。熱暴走ではなくてROMか何かのアクセスに問題があったのでしょう。

写真1.2　X68000用の68020ボード

話が脱線しましたが、たとえ32ビットプロセッサでも、16ビットバスのX68000のままでは性能が出ないという苦い経験があっただけに、内心、X68000の後継機種が、この安直な方式になるのではないか、と恐れていたのです。

この点では、X68030は25MHzにクロックアップされ、ちゃんと32ビットデータバスになっていましたから、清く正しい32ビットマシンとしての基準はクリアしていたわけです。もっとも、VRAMを含めI/O系はX68000の流用[*1]で、16ビットバスでアクセスも10MHz並みにやたら遅いということを後から知りましたので、かなり減点になりました。

ほかにも1600万色表示や、1024×768ドットの高解像度表示[*2]、CDクオリティの48kHz16ビットサンプリングのPCM音源など、あったらいいなと思える機能は挙げていけばきりがありません。しかし、考えてみると、もともとX68000は6万5000色表示、768×512ドットの解像度、15kHzの4ビットADPCM機能と、カタログスペックこそ劣るものの、標準でこれだけ装備していますから、私にとっては不満はありません。使用目的が偏っているからかもしれませんが、X68000にパワー不足以外の不満を感じていなかったので、ここらへんは目をつぶることができました。

むしろ、残念だったのはX68030にプロセッサの拡張性がなかったことです。拡張I/OスロットはあいかわらずX68000レベルの16ビット10MHzバスで、32ビット25MHzのプロセッサのパワーを引き出すことができません。

またまたMacintoshを引き合いに出して申し訳ないのですが、Macintoshには、PDS（Processor Direct Slot）といって、プロセッサのバスに直結したスロットがあり、グラフィックカードやアクセラレータカードなどを取り付

＊1
これなら、X68000用に68030と32ビットローカルバスを載せたボードを作れば、X68030と同じだと思った人もいるかもしれませんね。

＊2
これは改造可能です。第7章参照。

C O L U M N

X68000の5年伝説

X68000の発表された当時、本当かどうか真偽のほどは知りませんが、メーカーは5年間はX68000の基本仕様を変えないといったということで、ユーザーからおおいに支持されました。その頃は8ビットマシン大変革のときで、他の機種では半年や1年で大幅な機能強化がなされた新機種が出て、旧機種はどんどん陳腐化していく状況に私自身嫌気がさしていましたし、なによりもX68000のスペックは時代を先取りしていましたので、メーカーのいうように5年間は十分持つと思いました。

そして、変化の激しいコンピュータ業界にあって、X68000は本当に機種名とソフトウェアのバージョンアップしか行われずにきました。これは快挙だと思いますし、X68000は確かに5年は持ったと、今でも思っています。

けることが容易にできるようになっているのです。

X68030には、こういった拡張性が配慮されていません。幸い、プロセッサ自体はソケット取り付けになっているため、このソケット経由で無理やりプロセッサの信号を取り出すことができますが、あまりスマートなやり方とはいえないでしょう。拡張性に関しての配慮があれば、多少、基本スペックに不満があっても希望が持てる*[1]のですが、残念なところです。

こうやって見ると、X68030がいいマシンなのかダメなマシンなのか、わからなくなってしまいますが、先代のX68000が時代を先取りした、ずば抜けたマシンだっただけに、X68030の評価が辛口になっているといえるでしょう。

＊1
仮にX68030にPDS相当のスロットがあったとしても、アクセラレータカードなどが出る保証はありませんが、拡張の方法をメーカーがちゃんと用意していれば希望が持てます。

X68030と力の入ったソフトウェア

スペック的にはなんだかんだと不満があっても、やはり、プロセッサが16ビットの68000から32ビットの68EC030に変わり、クロックも上がったX68030は、確実にパワーアップしていました。

特にSX-WINDOWを使ってみると、それが実感できます。X68000でSX-WINDOWを使ったときは、ウィンドウをマウスでドラッグして移動するという基本的な操作でも、マウスボタンを離した後、ウィンドウが描画されるまでにワンテンポ待たされてイライラしました。SX-WINDOW利用者が少な

C O L U M N

でも、その次が期待外れでした。５年待ったのですから、次はもっと凄いものを期待するのが人情ですが、SUPER、XVI、compact XVI、そしてやっと、X68030です。これでは、他機種と同じです。いや、他機種だったら、X68000が発売された次の年にはもうやっていたことかもしれません。５年間モデルチェンジをしないということは、５年ごとには凄い機種が出ることを意味していると誰もが思っていたのに、実際には１年目のモデルチェンジが５年先に延びただけのようにも見えます。

X68030が発表されてから１年間がたちますが、いまだに新機種の噂は聞こえてきません。もしかしたら、X１からX68000になったように、今度はX68シリーズとはまったく違った、新しいアーキテクチャのマシンが出番を待っているのかもしれません。

いのは、アプリケーションや開発環境がないからというのが最も大きな原因でしょうが、私の場合は、このイライラのためにSX-WINDOWを積極的に使う気になれませんでした。10MHzのX68000*1では使う気にならなかったSX-WINDOWでしたが、X68030ではウィンドウの開閉も移動も実にスムーズで、非常に気持ちよく使えるのです。

また、6万5000色のグラフィックウィンドウや動画表示など、機能的にも非常におもしろくなっていました。X68030の開発は、ハードウェアよりもHuman68k ver 3とSX-WINDOWなどのソフトウエアのバージョンアップのほうに力が入っているといわれていましたが、確かにそう思わせるくらい、パワーアップしていました。そして、X68030は、それらのソフトウエアを余裕を持って動かすハードウェアパワーを持っているのです。

逆にいえば、SX-WINDOWのようにCPUパワーを食うプログラムを使ってみないと、X68030の恩恵があまり感じられないともいえます。

実際、フリーソフトウエアを中心として構成されているX68000の環境では、どれもストレスなく使えるよう非常に高速にチューンナップされたものばかりですから、X68030に移行しても目立って速くはなりません。むしろ、不具合が出ることのほうが多いのです。

不具合の筆頭は、画面のスクロールを高速にするために、X68000のCRTコントローラの機能の1つであるラスターコピーを直接使用した場合に画面にゴミが出るというものです。少しでも高速にするために、68000のスピードにあわせてCRTコントローラの制御を省略していたのが、68030の高速動作に引っかかってしまったのでした。

ちなみに、IOCSコールで用意されているラスターコピールーチンではこれへの対処がなされているために、IOCSコールを利用したプログラムでは不具合が起こりません。

このほかで問題になったのが自己書き換えプログラムです。自己書き換えというのは、動作しながら自分の実行すべき命令コードを書き換えていくような手法で、たとえば、条件分岐をしなくても実行内容を変えることができるため、高速化のテクニックの1つとして使われていたりします。ただ、命令コードをデータと見なして書き換えるため、68030のように命令キャッシュとデータキャッシュが分かれている*2と、命令キャッシュ上は書き換えられません。このため、前の命令のまま実行され、目的とする動作と異なってしまうのです。

X68030でプログラムするときは自己書き換えプログラムはやめましょうといわれていますが、X68000時代にはそんなことはいわれていなかったので、自己書き換えプログラムは結構あります。

＊1
もっとも、その頃の主力機種は16MHzになったX68000XVIですから、10MHzのマシンはターゲットになっていなかったのかもしれません。

＊2
ちなみに、インテルの80486は命令キャッシュとデータキャッシュの区別がありませんので、自己書き換えしても問題はないようです。

また、積極的に自己書き換えをしているわけではないのですが、結果的に同じ問題が起きるのが、lzxでした。lzxはプログラムを圧縮したまま実行できます。これは、実行時にlzxの展開ルーチンが呼び出されて、圧縮されたプログラム本体をメモリ上に展開した後、本来のプログラムを実行する仕組みになっているからです。しかし、この展開作業がまさに自己書き換えにあたるため、lzx化されたプログラムをX68030で動作させようとすると不具合が起こるのです。

こういった不具合はほかにもいくつかありましたが、すぐにパッチなどの対処がなされましたし、ほとんどのX68000時代のプログラムは問題なくX68030で使用することができたため、X68000からX68030への移行は非常にスムーズに行えました。

ここで、X68000からX68030に移行して劇的な速さを痛感した話を紹介しておきましょう。

やっぱりX68030は速い

X68030への移行が一段落し環境の整備も整った頃、浮動小数点演算プロセッサ68882を取り付けてみました。

X68030のコプロセッサはPLCC（Plastic Leaded Chip Carrier）という形状のもので、X68000のコプロセッサボードに使われていたPGA（Pin Grid Array）タイプは流用できません。かといって、純正品は５万円以上もするので、Macintosh専門店で１万円ちょっとで売っていたLC３&DUO用という33MHzの68882を購入しました。

さて、68882を取り付け、浮動小数点演算パッケージもFLOAT4.Xに取り替えました。こうなると、X68030の浮動小数点演算パワーを見てみたくなります。

友人に、レイトレーシングプログラムをCで起こし、24MHzにクロックアップしたX68000XVIでバリバリ使っているHat氏[*1]がいましたので、彼がそのプログラムを持ってわが家に遊びにきたときに、さっそくX68030でレイトレーシング[*2]を試してみました。

しかし、期待に反して、

そんなに速くないじゃない。

＊1
彼とは大学時代からの腐れ縁です。彼は8ビットパソコンの時代からレイトレーシングをやっていました。

＊2
浮動小数点演算を多用するので、68882のテストにはぴったりです。

X68000XVIのほうのプログラムは、GCCの吐き出す浮動小数点演算命令をFPPP.X[*1]というプログラムに通すことで、68881を直接駆動するようになっていました。これは、FLOAT3.X経由で68881を使うよりも数倍から数十倍高速です。

一方、X68030用のプログラムには、GCCの吐き出す浮動小数点命令をアセンブルできるアセンブラ[*2]がなかったため、浮動小数点演算命令を使わずにコンパイルしていました。このため、FLOAT4.X経由でコプロセッサを使わざるを得ません。一画面を描画する時間でいえば、X68000 XVI (24MHz) は7分、X68030は5分。その差は、わずかに1.4倍。ちなみに、10MHzのX68000は16分でした。これには2人ともガッカリしました。

しかし、簡単にはあきらめられません。さっそく、別の友人にあたってGCCの吐く浮動小数点演算命令を使えるようにするプログラムを入手しました。この結果、一画面を描画する時間は1分20秒にまで短縮されました。X68000XVI (24MHz) と比較して5倍以上、10MHzのX68000とでは12倍の差です。ベンチマークテストの結果に一喜一憂するのはくだらないことと思いつつも、やはり愛機が期待した性能を出せないと奮起してしまうのは、マイコン時代からの性(さが)でしょうか。

＊1
谷本孝浩さん作のフリーソフトウエア。

＊2
現在は、XC ver2.1 NET KITのASのほか、フリーソフトウエアのfas、hasが対応しています。

X68030のハードウェア

X68030で動かすソフトウェアの速さに満足した後は、またまたハードウェアへの興味がわいてきます。

最初にマザーボードをむき出しにした後は、ずっと、X68030のカバーは開けっぱなしのままなので、目の前で動いているハードウェアの挙動を見てみたくなります。こういう用途に威力を発揮するのが、ロジックアナライザです。ロジックアナライザは、ハードウェアの信号線にプローブを取り付け、信号レベルがHighかLowかをサンプリングする装置です。信号の組み合わせ条件を設定しておいて、それをトリガとして前後数千ステップをサンプリングできるので、ハードウェアの不具合の解析に威力を発揮します。

私の使っているロジックアナライザは、X68000に68020をつなぐときに必要に迫られて購入したDL8という製品（写真1.3）で7万円ちょっとという、ロジックアナライザとしては破格[*3]の値段のものです。ただし、たったの8チャンネルしか見ることができず、サンプリングクロックも最高80MHzですか

＊3
同じような位置付けの製品で、PA-200というものもあります。こちらは10万円をちょっと超えますが、100MHz16チャンネルか、200MHz8チャンネルで使えます。

ら、10MHzの68000で使う分には十分でしたが、25MHzの68030で使う場合には少々力不足です。しかし、まともなロジックアナライザは数百万円もしますから、これでもしかたがありません。

ちなみに、このDL８は、PC-9800シリーズをホストとして使うようになっているため、中古のPC-9801Uも購入しましたが、これはたったの１万円でした。

さて、実際にロジックアナライザを使ってX68030の信号を観察するわけですが、68030は68000と違ってPGAというタイプのLSIを使っているので、表から端子に触ることができません。このために、市販のエクステンション基板を改造して*1、信号を取り出すテスト基板を作りました（写真1.4）。

X68030のマザーボードから68030を抜いて、このテスト基板を取り付け、その上に68030を取り付けます。これでX68030のメモリアクセスについて、ROM、RAM、VRAMについて調べてみました。

＊1
このエクステンション基板は12×12列なので、そのままでは13×13列の68030には使えません。

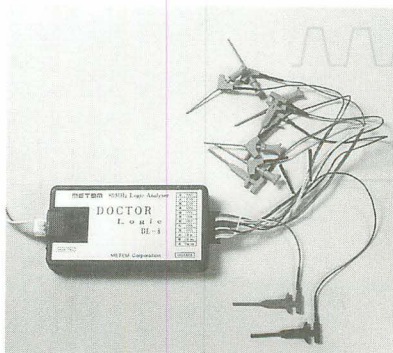



写真1.3　DL８の外観（左）と観測中の波形（右）

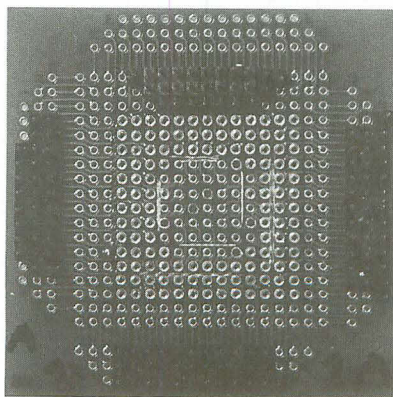
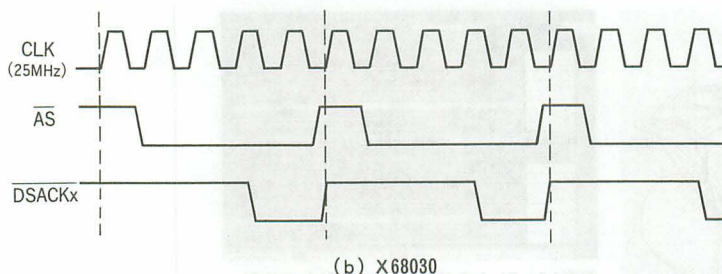
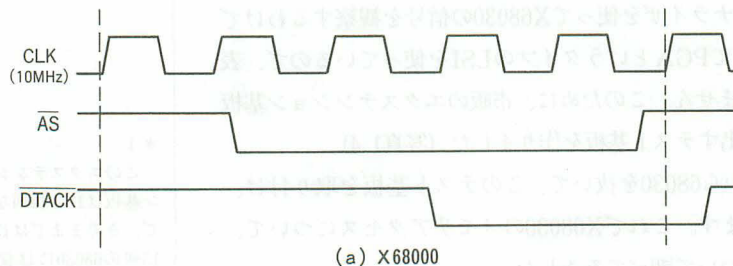



写真1.4　自作したエクステンション基板（左）と、取り付けて観測中の様子（右）

ROMアクセス

X68000とX68030のROMの違いは、バス幅とクロックだけと見ていいでしょう。図1.1.aにX68000のアクセスの様子を、図1.1.bにX68030のアクセスの様子を示します。所要クロック数は、68000で5クロック、68030でも5クロック*1。さらに、バス幅は68030が68000の2倍ですから、トータルとして68000の5倍といえます。

＊1
同じクロックでも1クロックの時間がX68030のほうが短いですから2.5倍速いといえます。

(a) X68000

(b) X68030

図1.1　ROMアクセスのタイミング

RAMアクセス

X68000、X68030のメインメモリはDRAMにより構成されています。DRAMは、図1.2のような内部構成になっており、1つのアドレス入力ピンをロウアドレスとカラムアドレスという2種類で時分割して使用するようになっています。そして、今、どちらのアドレスを与えているかをRAS（Row Address Strobe）、CAS（Column Address Strobe）という信号で指定するようになっています。このため、RAMのタイミングは多少ROMよりも複雑になります。

図1.3が10MHzのX68000のRAMのタイミングです。68000のバスサイクルはもともと最少でも４クロックなので、これはノーウェイト*1で動作しているといえます。

これに対し、X68030では25MHzという、さらに高速なクロックになり、68030自体のバスサイクルも３クロックですむようになったため、X68000時代のメモリ構成ではウェイトが入りまくってしまい、68030の性能を発揮できません。

このため、X68030ではスタティックカラムモードという特殊なアクセス*2方式が採用されています。スタティックカラムモードは、ロウアドレスが同じアドレスのアクセスの場合、最初の１回目は通常と同じようにロウアドレスとカラムアドレスを与えてアクセスしますが、２回目以降も同じロウアドレスの範囲になる場合、RAS信号を出しっぱなしにしておけば、ロウアドレスの指定を省略して、いきなりカラムアドレスを与えるだけで高速にアクセスできるというものです。

＊1
一部の拡張メモリボードでは、ウェイトが入るものもあります。

＊2
ほかにも高速ページモードやニブルモードといった特殊なアクセス方式がありますが、これはDRAMのタイプで決まっています。

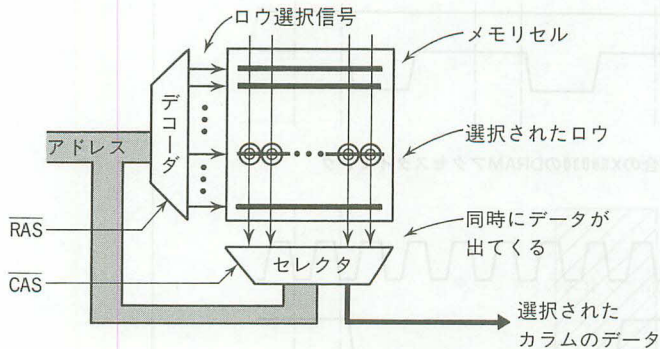



図1.2　DRAMの構造（概念図）

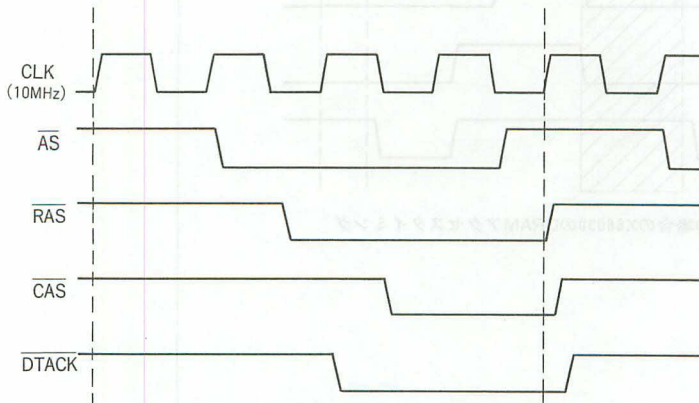



図1.3　X68000のDRAMアクセスのタイミング

X68030のメモリアクセスは、このスタティックカラムモードがあるために、アクセスのしかたによって、タイミングがかなり変化することになります。図1.4は、スタティックカラムモードが効くように、ある番地から連続してメモリアクセスした場合です。最初のアクセスには5クロックの時間がかかっていますが、次からは3クロックになっています。スタティックカラムモードが効いている場合はノーウェイトになっているといっていいでしょう。

＊1
命令実行やアドレス算出などの内部処理をしています。外部アクセスが遅ければ隠れてしまうが、アクセスが速いと見えてきます。

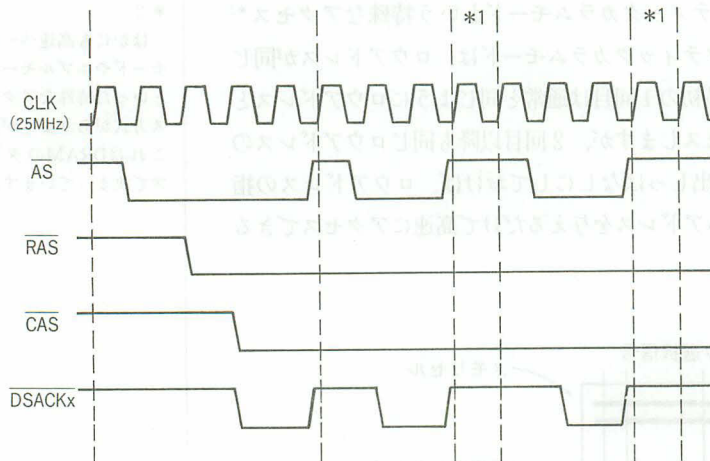



図1.4 連続アクセスの場合のX68030のDRAMアクセスタイミング

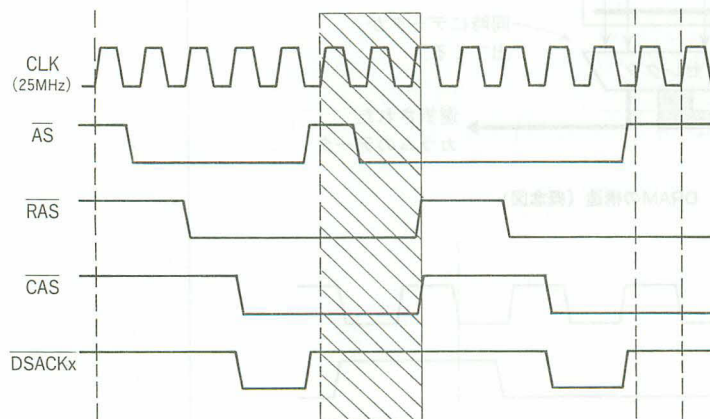



図1.5 非連続アクセスの場合のX68030のDRAMアクセスタイミング

逆に、スタティックカラムモードが効かなくなるように、わざとロウアドレスが変わるような番地にアクセスしたのが図1.5です。

メモリコントローラは、スタティックカラムモードでアクセスできることを期待して、前のアクセスが終わってもRASを出しっぱなしにしているために、次のアクセスに入ってから、図1.5の斜線部分のようにいったんRASを戻さなければならず、2クロックもよけいにかかって7クロックになっています。

また、X68030は、スタティックカラムモードを使わないようにする*1こともできます。この場合は、図1.6のようになり、コンスタントに5クロックかかります。

＊1
隠し機能のようですが、マザーボード上のある場所にジャンパー線を張ることで、スタティックカラムモードを切ることができます。後で詳しく説明します。

スタティックカラムモードは、メモリアクセスのパターンによっては効率が悪くなる可能性もありますが、普通、データはある程度まとまって配置されるので、トータル的に見ればスタティクカラムモードを使わないよりも、使ったほうがスピードが速くなるでしょう。いろいろなベンチマークプログラムを使ってみた感じでは、スタティックカラムモードを使った場合は、使わない場合に比べ、およそ20%くらい速いようです。

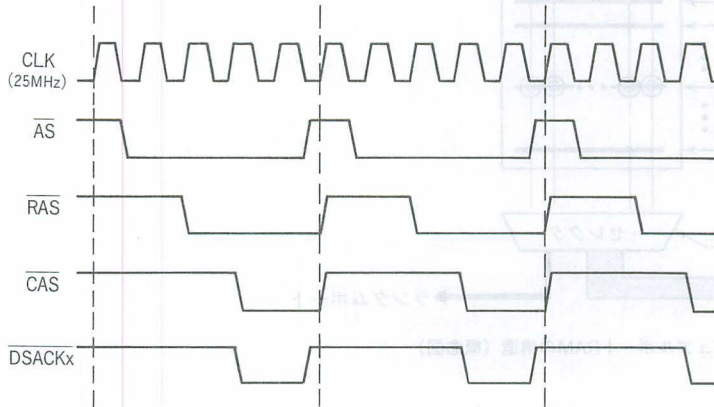



図1.6　スタティックカラムモードオフのときのX68030のDRAMアクセスタイミング

VRAMアクセス

X68000シリーズの表示系は、テキスト画面、グラフィック画面、スプライト画面がありますが、ここでは、テキスト画面とグラフィック画面を構成するVRAMについて説明します。

このVRAMには、デュアルポートRAMが使われており、図1.7のように通常のDRAMと同様にランダムアクセスが可能なポートに加え、任意のロウアドレスで指定された1行分のデータを連続してアクセスするためのシリアルポートがあります。

ランダムアクセスポートはプロセッサからのアクセスに、シリアルポートは画面表示するためのアクセスに使用されます。このため、表示のためのVRAMアクセスとプロセッサからのアクセスが基本的にぶつかることがありませんから、いつでも自由にVRAMをアクセスすることができます。

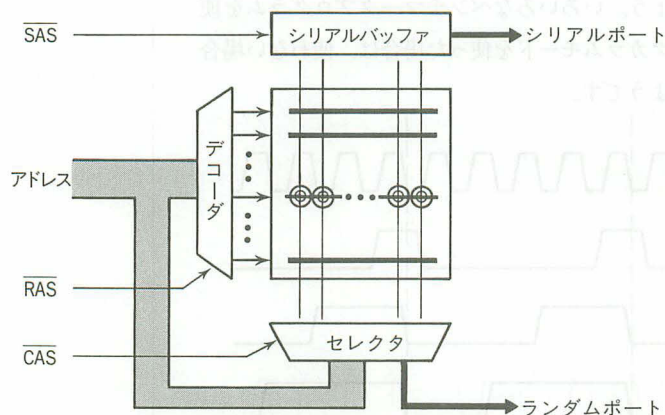



図1.7　デュアルポートRAMの構造（概念図）

COLUMN

VRAMアクセスのウェイト

MZ-80Kは、デュアルポートRAMなんて便利なものはなかったので、表示のためのVRAMアクセスとプロセッサとアクセスがぶつからないようにするため、ソフトウェアで垂直同期期間をチェックして、このときだけにアクセスを制限していました。これを無視してVRAMアクセスすると、画面にノイズが出てしまいました。ちなみに、名機APPLE IIは、普通のDRAMを使っていましたが、ハードウェアの作りが絶品で、アクセスがまったくぶつからないように工夫してありました。これには憧れたものです。

図1.8.aが、10MHzのX68000のVRAMアクセスの様子です。

X68030では、VRAMを含めI/O系はX68000の回路を流用しているようで、16ビットのままです。動作クロックは12.5MHzと若干上がっていますが、決して速くなったとはいえません。図1.8.bに、その様子を示します。1回のアクセスに実に10クロックかかっていることになります。

X68000との互換性を維持するために、VRAMはX68000と同じ構成をとらざるを得なかったのかもしれませんが、メインメモリは高速化にあわせて工夫されているだけに、VRAMアクセスのこの遅さは残念なところです。

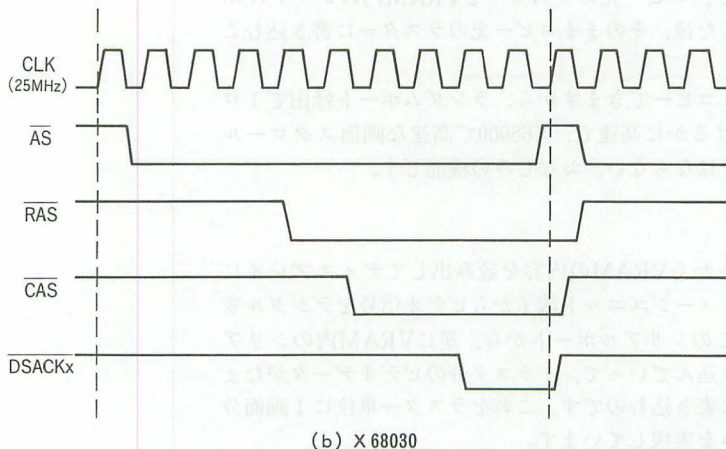
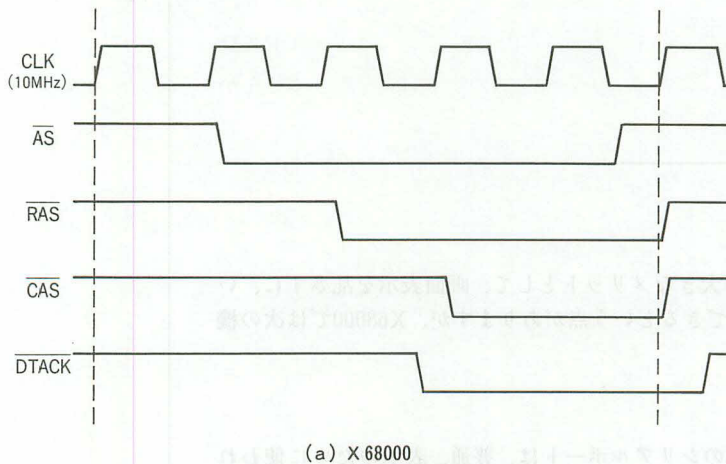



図1.8 VRAMのアクセスタイミング

VRAMは、メモリ容量的にも、メインメモリに比べ効率が非常に悪いものになっています。メインメモリを構成するDRAMは8個で4Mバイトなのに対し、VRAM*1は32個（テキストVRAM+グラフィックVRAM）使って、やっと1Mバイトしかありません。X68000の表示まわりのハードウェアは凝った作りになっていますから、互換をとるのも大変で、VRAMに使っているメモリチップ*2を安易に変えたりすることができないのでしょうが、画面表示は注目度が高い部分ですから、頑張って高速・ハイレゾ・フルカラー化を目指してほしかったところです。

＊1
256Kビットのデュアルポート RAM。

＊2
Macintoshなどは SIMMでVRAMを拡張できるようになっています。

COLUMN

VRAMの特殊機能

デュアルポートRAMの大きなメリットとして、画面表示を乱さずに、いつでもVRAMをアクセスできるという点がありますが、X68000では次の機能にも活用されています。

ラスターコピー

デュアルポートVRAMのシリアルポートは、普通、表示のために使われていますが、これを流用し、コピー元のラスターをVRAM内のシリアルポート用バッファにセットした後、そのままコピー先のラスターに書き込むことで実現しています。

ラスター単位でごっそりコピーできますから、ランダムポート経由で1ワードずつコピーするよりはるかに高速で、X68000で高速な画面スクロールを実現するためにはなくてはならない、おなじみの機能です。

ビデオ取り込み

普通は、シリアルポートからVRAMの内容を読み出してディスプレイに表示しているのですが、イメージユニット端子からビデオ信号をデジタル変換したデータをもらい、このシリアルポートから、逆にVRAM内のシリアルポート用バッファに書き込んでいって、1ラスタ分のビデオデータがたまったところでメモリセルに書き込むのです。これをラスター単位に1画面分繰り返してビデオ取り込みを実現しています。

これらは、デュアルポートRAMのチップが持つ、ハードウェアにべったり依存した機能ですから、互換を考えるとVRAMを安価なものに変えるといった仕様変更は、おいそれとはできないでしょう。

68030のパワー

さて、シャープの広告によれば、X68030の演算速度はX68000の2倍以上、DHRYSTONEベンチマークでX68000比4.3倍、16MHzのX68000XVI比で2.4倍となっています。バスが16ビットから32ビットになって2倍、クロックが10MHzから25MHzになって2.5倍、あわせて5倍を期待したいところですが、さすがにそれは無理のようです。しかし、この数字は、まあ、いい線をいっていると見ていいでしょう。

X68000XVIが登場したときはクロック比でX68000の1.6倍しかなかったにもかかわらず、IOCS.Xなどのシステムソフトウェアの改良により画面表示が高速化したということで、体感速度は2倍と広告していたのですから、これに比べればだいぶ控えめにはなっています。実際のアプリケーションにおいては画面表示やディスクアクセスなど、マシン全体の性能が効いてくるため、I/O系にほとんど改良を加えなかったX68030ですから、速度について大きなことはいえないのかもしれません。それでも、X68030は確実に高速化しました。実際のところ、プロセッサの違いがどう性能に影響してくるのか、もう少し突っ込んで説明しましょう。

C O L U M N

68EC030

ここでは、あえてX68030のプロセッサを68030として説明してきましたが、実際には68030からMMU（Memory Management Unit）機能を省いたサブセットである68EC030がX68030には搭載されています。ちなみに〝EC〟を、私は最初エコノミーのことかと思っていましたが、Embedded Controllerの略だそうで、制御用の分野に特化したプロセッサという意味だそうです。

MMUがないこと以外はソフトウェア的にも同じといってよく、Human68kで使う分には支障がありません。逆に、仮想記憶などMMUを必要とする機能に関しては、今後、Human68kがサポートする見通しもほとんどないわけですから、ちょっと寂しいところです。

ちなみに、68EC030のピン数は68030よりも少ないのですが、X68030のマザーボードには68030相当のICソケットが使用されているため、68EC030を68030に交換することが可能です。

68000と68030の違いは以下の点にあります。

1）バス幅
2）クロック
3）オンチップキャッシュ（命令用、データ用でそれぞれ256バイト）
4）プロセッサ自体の改良
5）68882のコプロセッサ化

1）バス幅

68000はデータバスが16ビットであるのに対し、68030は32ビットです。これは、単純に考えると、1回のバスアクセスあたり2倍のデータをアクセスできるということになります。しかし、つねに2倍かというとそうはいきません。

32ビットのロングワードデータのアクセスを考えると、68000では間違いなく、2回のバスアクセスが必要になります。一方、68030も、データがロングワード境界、つまり、0、4、8、……といった4の倍数にあたるアドレスにあれば1回ですみますが、そうでないときには、やはり2回かかります。また、8ビットや16ビットのデータをアクセスする場合、バス幅は関係なくなってしまいます。

XCやGCCなどのCコンパイラで作ったプログラムは、基本となるintがロングワードなので、その差が結構出るかもしれませんが、速度重視のため、マシン語でバリバリにチューンナップされたフリーソフトウェアではわざわざ不利なロングワードアクセスを使わずに、68000の得意な16ビットのワードアクセスを使ってプログラムされているでしょう。皮肉なことに、スピードアップの期待されるプログラムほど、68030の32ビットバスの恩恵が得られないわけです。

もっとも、キャッシュがかかわってくると話は変わってきます。キャッシュをオンにするとメモリアクセスは32ビット単位で行われるようになるので、8ビットや16ビット単位でアクセスするようなプログラムでも連続したアクセスなら、2回目のアクセスはキャッシュ上ですむ[*1]ため、無駄が少なくなり、32ビットバスの効果が出てきます。

なお、X68030では、VRAMを含めI/O系がX68000と同じ16ビットのままであるうえ、ハードウェアでCIIN（Cache Inhibit IN）信号が必ず返されるようになっており、キャッシュの使用が禁止されます。このため、I/O系では68030でバス幅が32ビットになった恩恵はまったくありません。

＊1
ただし、これはメモリ読み込みだけです。書き込みは実際にアクセスが起こるので駄目ですが、普通は書き込みだけのプログラムというのはないので、キャッシュによる高速化を期待できるでしょう。

2）クロック

X68000のクロックが10MHz、X68000XVIで16MHzであったのに対し、X68030では25MHzに引き上げられました。クロック周波数が上がっても、メモリアクセスにウェイトが入りまくって10MHz並みのレスポンスしか得られなければ*1、クロックアップの効果は期待できませんが、X68030のメインメモリはスタティックカラムモードが有効に機能した場合、ほぼノーウェイトで動作するので、25MHzに引き上げられたクロックに見合った高速動作が実現されていると見ていいでしょう。

さらに、ここでもキャッシュがおおいに関係してきます。いったんキャッシュに読み込まれてしまえば、同じメモリからの読み出しに関しては内部処理だけですみます。メモリアクセスの場合、バスサイクルに最低でも3クロックかかりますが、キャッシュからなら1クロックで読み出せますから、メモリアクセスの遅さが足を引っ張ることがなくなります。

もっともX68030の場合、キャッシュオフでも10MHzのX68000よりは速く、24MHzにクロックアップしたX68000XVIくらいの速度は出ますので、だいたいクロック比に見合った高速化は達成できているといえるでしょう。

なお、I/O系のアクセスではウェイトが入りまくる*2し、キャッシュも効かないので、クロック25MHzの恩恵はこれまたほとんどありません。画面表示が大半を占めるプログラムでは、残念ながら、顕著な効果は出ません。

＊1
X68000互換のI/O系は、まさにそういう状況になっています。

＊2
VRAMアクセスの10クロックという数字から見てI/O系は12.5MHzで動いているようです。

3）オンチップキャッシュ

X68030の場合、アプリケーションによっては、キャッシュオフで動かした場合とキャッシュオンで動かした場合では2倍以上の性能差があります。一度、キャッシュオンの速度に慣れてしまうと、キャッシュオフの動作がまだるっこしく感じられてしまいます。このため、キャッシュが高速化の源泉であると錯覚しがちですが、もともと、キャッシュは高速なプロセッサと遅いメモリとのギャップを埋めるためのものにすぎません。

最近の高速プロセッサでは、数Kから数十Kバイトのキャッシュを持っているのに対し、68030のキャッシュは命令キャッシュ、データキャッシュのそれぞれ256バイト*3しかありません。ただし、仮に68030が数Kバイトのキャッシュを持っていたとしても、プロセッサ自体の処理速度が頭打ちになるので、全体としてはそんなには速くはならないでしょう。キャッシュは、バス幅やクロックのところで説明したように、プロセッサ本来

＊3
68030発表当時の半導体テクノロジーとしてはこれくらいが限界だったのでしょう。ちなみに、80386はキャッシュを積んでいません。

の性能を引き出すために存在しているわけです。

4）プロセッサ自体の改良

68000と68030とを比べると、プロセッサ自体も改良されており、バスサイクルが4クロックから3クロックに短縮されています。

さらに、68030では、命令の実行とある程度オーバーラップ*1して、命令やデータのアクセスを行えるようになっていますので、相対的に一命令の実行に要する時間が短くなっています。

＊1
このため、68030では命令キャッシュケース、ノーキャッシュケースに分かれ、そのなかでも細かく条件が分かれるので、命令の実行時間の算出が非常に面倒になっています。

このほか、個々の命令実行についても改良が行われています。68000にはなかった32ビットの乗除算命令も装備しましたし、シフト処理などは68000ではシフト数によって実行時間が増えていましたが、68030ではシフト数によらず一定時間で完了します。

ただし、68020をX68000に載せてもほとんど速くならなかったように、実際のアプリケーションの使用においてはプロセッサ自体の改良の恩恵は目立つほど多くはありません。

5）68882のコプロセッサ化

ある意味では、これがX68030の最大の性能アップポイントといってもいいでしょう。浮動小数点演算能力にかぎっていえば、十数倍という大幅な性能アップになっています。前に紹介したレイトレーシングプログラムのような、浮動小数点演算を多用するプログラムでは、その威力を発揮してくれます。

これは、68030が68882をコプロセッサとして接続するようになったことによる性能アップです。実際の浮動小数点演算は68882が行うわけですが、68030との間でコプロセッサインタフェースというハードウエア機構によって自動的に処理されるので、ソフトウェアからはあたかも68030に浮動小数点数レジスタと浮動小数点演算命令が備わったかのように扱うことができます。

X68000の数値演算プロセッサボードやX68000XVIに搭載された数値演算プロセッサにも、68881*2が使われていますが、68000自体がコプロセッサをつなぐことができるようにできていないので、I/Oとしてつながっているだけでコプロセッサとしては機能しません。このため、データを与えて浮動小数点演算を実行させるなどの、68030であればハードウェアとしてコプロセッサインタフェースが面倒を見てくれる処理を、I/Oへの入出力の形ですべてプログラムしてやらなければならないのです。その分、

＊2
X68030に使われている68882は68881の改良版ですが、性能差はそんなにありません。

オーバーヘッドが大きくなり、スピードが上がりません。

たとえば、浮動小数点数レジスタfp０の内容を２倍するのに、68030では、

```
fmul.d #2.0,fp0
```

このように１行ですむ乗算命令も、68000では次のようなプログラム[*1]にしなければなりません。

```
loop1:move.w #$5423,$00E9E00A       ←fmul.d相当の命令
      cmpi.w #$9608,$00E9E000       ←ステータスチェック
      bne.w  loop1
      move.l #$40000000,$00E9E010   ←あわせて
      move.l #$00000000,$00E9E010     浮動小数点の2.0
loop2:tst.w $00E9E000               ←ステータスチェック
      bmi.s  loop2
```

もっとも、Human68k標準の浮動小数点演算パッケージであるFLOAT4.X[*2]を通して使う場合は、FLOAT4.X自体の呼び出しオーバーヘッドが大きいために、68882のコプロセッサ化の恩恵はほとんど期待できません。あくまで、浮動小数点演算命令を直接使った専用プログラムでなければ駄目です。しかし、性能差がこれほどあるなら、X68030専用のプログラムになってしまってもやむを得ないでしょう。

以上、見てきたように、68030は68000から多くの点で進歩し、処理性能もそれなりに上がっています。ほんの数年前まではワークステーション[*3]に使われていたチップなのですから、それがパソコンに使われているというだけでも凄いことといえるかもしれません。

X68000のキャッチフレーズは「パーソナルワークステーション」でしたが、68030を心臓部に持つことでX68030はハードウェア的にはワークステーション並みになったわけです。

しかし、心臓部に68030を使ったことがX68030のパワーの源であると同時に、限界でもあります。仮に68030をクロックアップした後継機種が出たとしても、68030自体の最高クロックは50MHzですから、X68030L[*4]止まりです。

今やワークステーションは、68030アーキテクチャに見切りをつけてRISC

＊１　このプログラムはFPPP.Xが生成したものを読みやすいように修正したものです。

＊２　FLOAT4.Xパッケージで数値演算を統一的に扱うことにより、演算プロセッサの有無などでアプリケーションを使い分けなくてもすむという今の方式は、当初、私もいいアイデアだと思っていました。

＊３　SUN３やソニーのNEWSなど、ほとんどのメーカーで68020や68030が使われていました。

＊４　Lはローマ数字の50。

プロセッサに移行してしまいましたし、残念ながら、68030は旬を過ぎ、すでに速度的に一世代から二世代前のプロセッサになってしまったといえます。

Voice of Users

このコーナーは"040turboの基板配布"に参加して、すでにX68030上で68040の世界を体験している方の感想を集めたものです。

骨までしゃぶれる68040

68000から見れば、68030は十分高性能ではあるのですが、やはり68040という超高性能なCPUを一方で知ってしまっている以上、「いつかは68040を骨までしゃぶれるマシンがほしい」と思っていました。そこへ040turboの登場です。これはもう、040turboの配付を申し込まずにはいられませんでした。ほかの多くのX68030ユーザーも当然同じ思いだと思っていました。

しかし、僕の行っているネットに040turboの紹介の記事を転載しても、なぜかほとんど反応がありませんでしたし、いまだに040turboの知名度は低いように思います。なぜなんでしょうねぇ。もっとも、あまり有名になってしまうと、BEEPsさんも対応するのが大変になってしまうから、今くらいがちょうどいいのかもしれませんが。

さて、最近ちょうどいい具合にモトローラから68060が出てきたみたいですね。というわけで、今度は060turboをお願いしますよ。(^_^) ＞BEEPsさん

（文●なっち（湯浅夏樹）NIFTY-Serve KHF03720）

68040のパワー

X68030の登場前から、68000はもとより、68020、68030、68040のユーザーズマニュアルをすべて揃えて眺めていましたから、X68030のハードウェアについて理解が深まれば深まるほど、ますます、これが68040だったら、という思いがつのります。

68040のほうが68030より新しいのですから、性能がいいのは当たり前ですが、68040のハードウェアの高速化に対する徹底的な改良はかなりのものです。

68000から68020、68030への改良というのは、どちらかというと互換性重視のゆっくりした性能アップ*1でしたが、RISCプロセッサの台頭を受けて、そんな悠長なことはやっていられなくなり、68040では互換性をバッサリ切り捨ててまでも性能アップを図っています。

68040では、実際にどんな高速化のための工夫がなされているのか、68000と68030の違いを見たように、68030と比較しつつ、次の点について説明しましょう。

1）バスアクセス方式
2）クロック
3）オンチップキャッシュ（命令用、データ用でそれぞれ4Kバイト）
4）プロセッサ自体の改良
5）浮動小数点演算機能の内蔵と高速化

*1
ワークステーションでの採用実績ナンバーワンにあぐらをかいていたのでしょう。

1）バスアクセス方式

68000と68030とではバス幅が違っていましたが、68030と68040は同じ32ビットバスです。しかし、バスアクセスの方式が大幅に変わりました。68030は68000とほぼ同じバスアクセス方式で、プロセッサからはAS（Address Strobe）信号を出し、デバイス側はデータを処理したらDSACK（Data Size ACKnowledge）信号で応答するというハンドシェークで行われます。このため、基本的にプロセッサとデバイス側の動作クロックが一致している必要はありません。「非同期式のバス」と呼ばれているゆえんです。

さて、68000から68030に受け継がれた非同期式バスですが、状況が変わってきました。プロセッサの速度に対し、非同期式バスではメモリアク

セスの遅さが目立つようになったのです。1クロックでも速くアクセスしたくなると、どうしても別のバスアクセス方式が必要になります。ハンドシェークの発想では、デバイス側は図1.9.(a)のようにデータが揃ったらアクノリッジを返しますが、プロセッサはアクノリッジを認識した後、次のクロックでデータを取り込むので、この間の1クロックが無駄になります。悠長にデータが揃うのを待ってアクノリッジを返していたのでは時間の無駄というわけで、図1.9.(b)のようにデータが揃うのを見越して、アクノリッジの先出し*1が行われます。

しかし、いかに性能を引き出すためとはいえ、先行してアクノリッジを返すというのは苦肉の策*2という感じがします。それでも、プロセッサがアクノリッジを確認する過程が入りますから、最短でも3クロックかかることになります。

＊1
X68000のメモリもこうなっています。非同期式バスを信じてクロックアップすると、これでハマることになります。

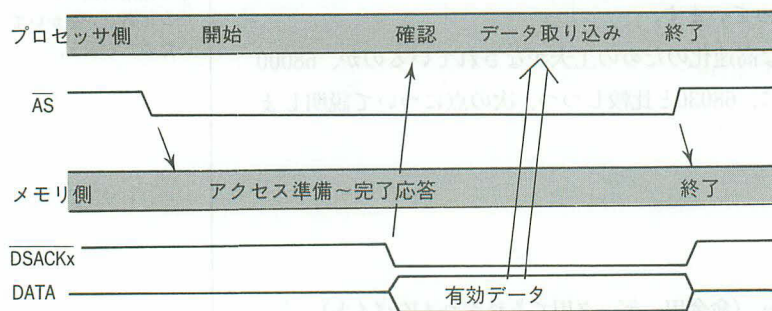
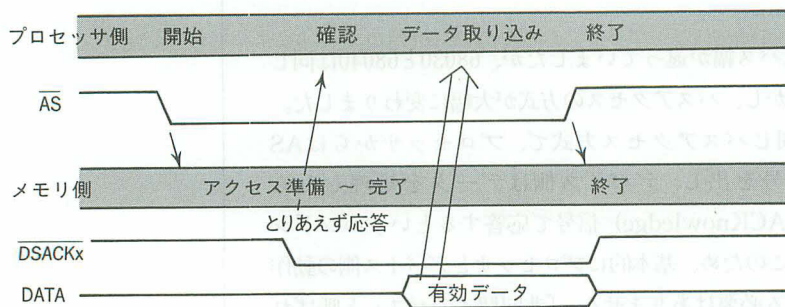
＊2
この方式は、68030のユーザーズマニュアルでもDSACK信号との同期動作として紹介されていますから、いちおう、正規の方式といってもいいかもしれませんが。

(a) 本来のハンドシェーク動作

(b) 応答の先出しによる動作

図1.9　68030のバスアクセスタイミング

これに対し68040では、図1.10のようにプロセッサからのTS(Transfer Start）信号に対し、デバイス側からのTA（Transfer Acknowledge）信号で応答するようになっています。68040は、クロックの立ち上がりエッジで、TA信号の確認とデータの取り込みを行うので、最小２クロックでアクセスが完了します。

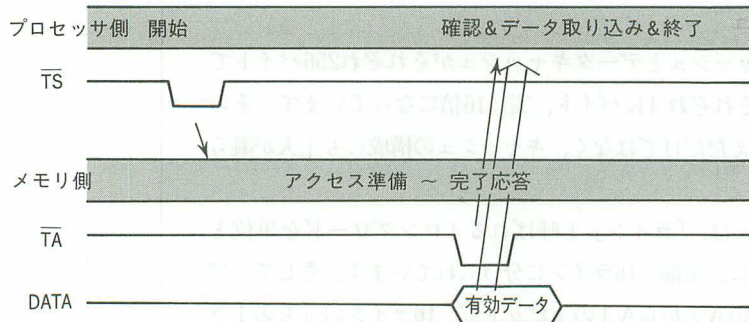



図1.10　68040のバスアクセスタイミング

モトローラの８ビット時代のプロセッサである6800は、クロック（Eクロック）を基準としてメモリアクセスを行う同期式バスでしたが、68000の非同期式を経て、またシンプルな同期式に戻ったというわけです。

ついでにいうと、ダイナミックバスサイジング機構は性能向上の邪魔物と判断されたのか、68040ではなくなってしまいました。確かに、バスアクセス方式自体が大幅に変更されているし、今さら16ビットバスで使うこともないでしょうから、この複雑なメカニズムを装備するくらいなら、シンプルなバスアクセス方式１本に絞って、もっと他の回路に力を注ぐほうが得策なのでしょう。しかし、このために、68040をX68030に載せるのは大変な作業になってしまいました。

２）クロック

25MHzの68030は本当に25MHzで動作していますが、25MHzの68040の場合はバスアクセスが25MHzというだけで、内部はその倍の50MHzで動作しています。実際、このために68040には、25MHzのBCLK（Bus CLocK）と、50MHzのPCLK（Processor CLocK）という２つの信号を入れなければなりません。

この発想は、80486DX 2 [*1]と同じといってもいいでしょう。もっとも、80486DX 2は内部にクロックダブラを持っていて、外部からはバスクロックを与えるだけですんでいます。

キャッシュが効いて内部処理だけで考えれば、単純にいっても68040は68030の2倍の速度が出せるわけです。

*1 33MHzのバスクロックを与えている80486DX2を「66MHz」というのなら、68040も「50MHz」といっていいでしょう。

3）オンチップキャッシュ

68030は、命令キャッシュとデータキャッシュがそれぞれ256バイトでしたが、68040ではそれぞれ4Kバイト、実に16倍になっています。そのうえ、ただ容量が増えただけではなく、キャッシュの構成にも工夫が凝らされています。

68030のキャッシュは、「ライン」と呼ばれる4ロングワードを単位として、図1.11のように、全部で16ラインに分けられています。そして、アクセスするアドレスのA7からA4の4ビットで、16ラインのうちの1ラインを直接選択する「ダイレクトマップ」と呼ばれる方式になっています。このため、アドレスのA7からA4のパターンが同じもの、たとえば$00000000というアドレスと$FFFFFF0Fというアドレスのデータは同じラインになりますから、同時にキャッシュ上に置いておくことができません。

普通は連続したアクセスになるので、そう頻繁には問題になりませんが、仮にこの2つのアドレスを交互にアクセスすると、おたがいに相手をキャッシュから追い出すことになりますので、効率がガタ落ちになります。

これに対し68040のキャッシュは、図1.12のように、64ラインのキャッ

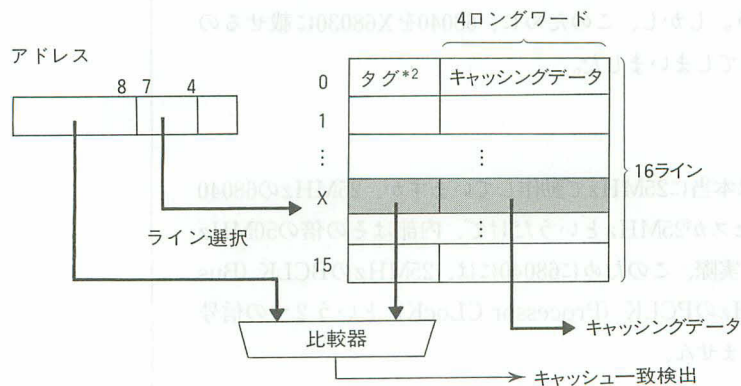



図1.11　68030のキャッシュの構造

*2 タグには、キャッシングされたデータの存在するアドレスの上位ビットが格納される。

シュを4本独立して持つ「4ウェイセットアソシエイティブ」と呼ばれる方式になっています。このため、同じラインにあたるアドレスでも4つまでは同時にキャッシュに置いておけるのです。キャッシュサイズの増加とあいまって、かなりの効果が期待できます。

さらに、68040のデータキャッシュは、データ読み込み時だけでなく、書き込み時にもキャッシュ効果を生かす「コピーバックモード」と呼ばれる、キャッシュ動作モードを持っています。

68030では、メモリ書き込みのときは実際のメモリアクセスが行われますが、68040のコピーバックモードでは、書き込み時もキャッシュ上のデータしか更新されません。実際のメモリ書き込みを必要なときだけ*1にかぎることができるので、さらに性能向上が期待できます。

*1
キャッシュラインの4本すべてが埋まって別のデータをキャッシングするときなどに、4セットのうちのどれかが疑似的な乱数で選択されて書き戻されます。

4）プロセッサ自体の改良

68040ではバスアクセスが最小2クロックに短縮され、命令の実行自体もパイプライン処理により大半の命令が1クロックで処理されます。実際、

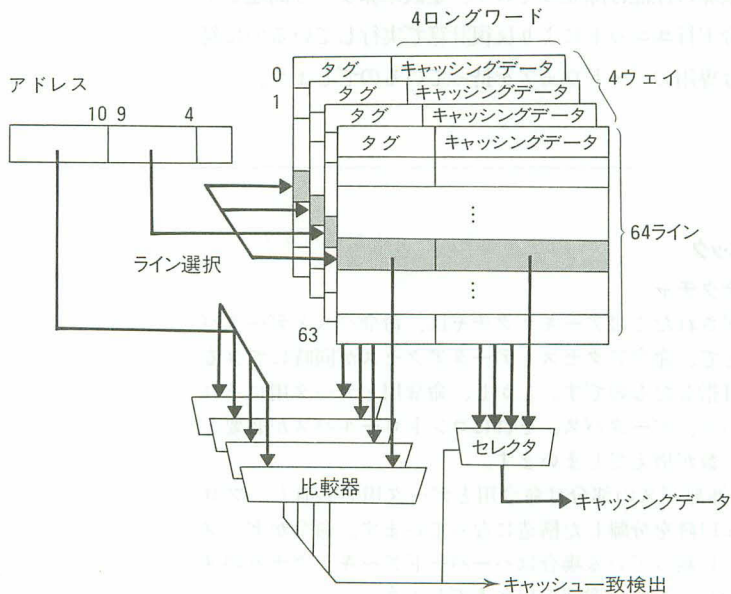



図1.12　68040のキャッシュの構造

データキャッシュをオフにした状態での68030と68040のプログラムの実行状況をロジックアナライザで見てみると、68030がアクセスしては考え、アクセスしては考え、という感じで動いているのに対し、68040はほぼ間断なくアクセスしています。メモリアクセスの間に、ほとんどの処理が完了してしまうからでしょう。

このほか、内部ハーバードアーキテクチャや、条件分岐のプリフェッチ、3段のライトバックバッファなど、高速化のための工夫が随所に凝らされており、マニュアルを読んでいるだけでワクワクしてきます。

5）浮動小数点演算機能の内蔵と高速化

68030での浮動小数点演算プロセッサの威力を説明しましたが、68040ではこれが内蔵されました。そして、コプロセッサインタフェースがなくなり、オーバーヘッドが減ったことやクロックが上がったことにより、さらに性能が向上しました。加減算および乗算は、5倍以上になっています。

特に、浮動小数点乗算の性能は際立っており、整数乗算よりも高速です。整数乗算が通常の命令実行ユニットにより反復計算で実行しているのに対し、浮動小数点乗算は専用のハードウェアを持っているのでしょう。

C O L U M N

68040の高速化のテクニック

内部ハーバードアーキテクチャ

ハーバード大学で考案されたこのアーキテクチャは、命令バスとデータバスを完全に分離することで、命令アクセスとデータアクセスが同時にできるようにし、高速動作を目指したものです。しかし、命令用とデータ用にそれぞれ独立したアドレスバス、データバス、それにコントロールバスが必要となり、プロセッサのピン数が増えてしまいます。

このため、68040では外部バスの部分は命令用とデータ用で共有し、プロセッサ内部のキャッシュ以降を分離した構造になっています。命令かデータのどちらかがキャッシュに載っている場合はハーバードアーキテクチャのメリットが生きてきますから、うまい選択といえるでしょう。

条件分岐のプリフェッチ

プリフェッチとは先読みのことです。68000や68030もある程度先読みはしていましたが、単純に先読みしているだけでは分岐命令が入ると先読みが無駄になり、あらためて分岐先の命令の読み込みをしなければなりません。

そのかわり、68882にあった、sin、cosといった超越関数の演算機能はバッサリ削られ、浮動小数点数の四則演算機能と平方根くらいになってしまいました。68040のユーザーズマニュアルでは、68882で実行するより68040のソフトウェアで超越関数をエミュレートするほうが速いと書かれていますが、確かに、このずば抜けた速さの乗算機能をもってすれば可能なのでしょう。

これらのさまざまな改良によって、68040は68030よりはるかに速いチップに仕上がっています。チップ単体での性能としては、68030が平均3～5 MIPS (Million Instruction Per Second) 程度*1であったのに対し、68040は20 MIPSを超えるといわれています。ちなみに、68000は1 MIPSに届くかどうかといった性能でした。

もっとも、最新のRISCプロセッサは、さらに改良が加えられ性能もアップし、100MIPSを超えるようになっています。それに比べると68040ももはや高速とはいえませんが、現時点では68000アーキテクチャの最高峰*2といえるでしょう。

＊1
1秒間に何回命令を実行できるかを表します。実際にはメモリアクセスの割合で大幅に変わるので、あまりあてにはなりません。

＊2
次の68060は100MIPSを超えるそうですが、詳しいことは知りません。68040ピンコンパチとも聞きますので期待しています。

COLUMN

これに対し68040では、プリフェッチ部で分岐命令を認識して、分岐先の命令を先読みするようになっています。さらに、条件分岐の場合は、条件が成立するかどうか実行されるまで不明なため、次の命令と分岐先の命令と両方を先読みするという念の入れようです。

X68000の自己書き換えプログラムでプリフェッチを回避するために、分岐先の命令を書き換えるというテクニックがありますが、68040のプリフェッチは賢いために、このテクニックは使えなくなりました。

ライトバックバッファ

命令の実行結果がメモリ書き込みの場合、命令実行部がいちいち付き合っているのは無駄ですから、ライトバックバッファに書き込んで、後は書き込み専用部に任せるようになっています。

68030では、このライトバックバッファが1段のため、メモリ書き込みが続くと結局ロックしてしまいます。

これに対し、68040は3段に増えていますから、効率が上がると期待できます。

X68000からX68030になるだけでも、結構な手間のかかる変更だったのでしょうが、68030はすでに終息しているのです。どうせなら、一気に68040まで進化してほしかったものです。

Voice of Users

やっぱり快速FPU

68030を040turboに替えて何がよかったか？

それは浮動小数点の演算が速くなったことですね。X68030に載せる68882もかなり速いのですが、68040の内蔵FPUはもっと高速です。フラクタル図形の表示プログラムを書いてみたのですが、68000 10MHzでは専用の演算ルーチンをアセンブラで作っても、表示するのに一晩以上かかっていたものが、68040なら分単位で処理できるので、とっても気分がいいですね。

68882も速かったけど、あんまり印象に残っていないです。

ただ、この040turboのスピードに慣れてしまうと、10MHzでX68000を使っているユーザーの気持ちを忘れちゃいますね。040turboで普通の速度で動くフリーソフトを作っても、普通のX68000では当然のことながら、すごく遅いので、最近はまったく相手にしてくれません（笑）。

やっぱり、プログラム書くときに実行速度を気にしなくてもいいのはストレスがたまらなくていいですよ。

（文●タタエル　（飯島匡史）NIFTY-Serve TCC01360）

第 2 章

040turbo計画

やっぱり68040がいい

X68000の後継機種にはなんとかあこがれの68040を、と期待していたのですが、残念ながら、68030でした。もっとも、X68000に無理やり載せた68020(「68020 on X68000」）は10MHzの16ビットバスが足を引っ張って性能がまったく発揮できませんでしたが、X68030はしっかり32ビット化してクロックも上がったので、相応の性能にはなっています。この点は、さすがといえます。

望んでいた32ビットマシンになったのだから、これで満足しなければいけないのかもしれませんが、なまじ68040のマニュアルを見てしまっただけに、

これが68040だったら

という思いはつのるばかりです。68030と68040のマニュアルを読み比べる毎日が続きました。

68000シリーズは、表2.1のように、10の位の偶数番号で世代が分かれており、この世代の境でハードウェア的に大きな変更があるのです。

「68020 on X68000」は、参考回路があったということもありますが、68020自体が、バス幅の違いはあるものの、68000のハードウェアにアクセスできるよう、ある程度の互換性を持っていましたので、比較的簡単な回路で実現することができました。しかし、68040にはそんな配慮はありません。趣味のハードウェア工作[*1]ではすみそうにありません。

考えれば考えるほど、68040は個人の手に負えるチップではないなあという

＊1
日曜大工のようなものです。

表2.1　68000シリーズの世代

世代	型番	特徴
第１世代	68000	元祖
	68008	68000のデータバスを８ビットにしたもの
	68010	68000の問題点を修正したもの
第２世代	68020	データバス、アドレスバスともに32ビットに拡張 256バイトの命令キャッシュ搭載
	68030	68020のマイナーチェンジ。MMUを内蔵 命令／データキャッシュそれぞれ256バイト搭載
第３世代	68040	あらゆる点が大幅変更。浮動小数点演算プロセッサ内蔵。命令／データキャッシュそれぞれ4Kバイト搭載

気になってきます。

しかし、当分X68040が出る気配はありません。とりあえず、X68030に思いとどまっていましたが、他機種への浮気心がわいてきます。

せっかくX68030を買ったんだ、ダメでもともと、とにかくやってみよう。

そして、1993年の５月、ゴールデンウィークの休みを利用して、この思いを実現すべく「68040 on X68030」の具体的な検討を開始しました。

ハードウェアの違い

とにかく、68030と68040ではハードウェアがかなり違いますから、これを克服しなければなりません。68040を68030のバスにつなげられるようにする変換回路[*1]を作るため、２つのプロセッサのバスアクセスのタイミングチャートを眺めながら、どうやったらいいか頭を悩ませることになります。

主なポイントは次のようなものでした。

1）非同期式のバスアクセスへの対応
2）ダイナミックバスサイジングへの対応
3）バスアービトレーション機能への対応

これらを解決していかなければなりません。

以下、この部分について、ちょっと説明しましょう。回路の詳細については付録の取扱説明書でさらに詳細に説明していますので、興味のある方はそちらもあわせてご覧ください。

1）非同期式のバスアクセスへの対応

X68030のバスは68030の非同期式のバスアクセスにあわせて設計されていますが、68040のバスは性能追求のために同期式のバスアクセスになりました。両者は信号の名前も動作もまるで違っていますから、68040の信号と68030の信号を変換回路で作り出してやらなければなりません。

変換回路の説明の前に、ここで68030のバスサイクルと68040のバスサイクルについて説明しておきましょう。

＊1
モトローラの「68040デザイナーズハンドブック」に、68030に68040を載せるための変換回路の参考例が載っていましたが、複雑な作りになっていたので、これを参考にするのはあきらめ、一から考えました。

●68030のバスサイクル

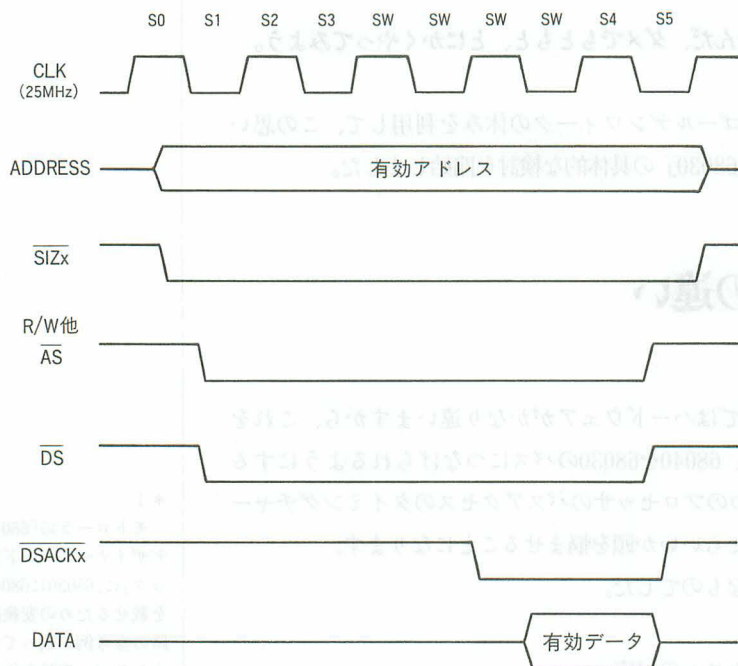
図2.1に2ウェイト[*1]で動作した場合のタイミングチャートを示します。説明のため、実際の信号名とはちょっと変えてあります。

＊1
ROMやスタティックカラムモードオフの場合のDRAMアクセスタイミングです。

図2.1　68030のバスサイクル

1つのバスサイクルは、1クロックの半分の時間を基準として、各ステートごとに次のように動作が定義されています。

ステート0(S0) プロセッサは、アクセスすべきアドレスとともに、リード／ライトを識別するR/W信号などを出力します。

ステート1(S1) アドレスが有効であることを示すAS (Address Strobe) 信号が出力されます。ちなみに、この信号も含め、多くの信号は、Low側で有効、High側が無効を示す負論理信号です。68000系では、信号が有効になることを「アサート」、信号が無効になることを「ネゲート」と呼んでいますので、以降はこの名称を使います。

ステート2(S2) メモリやI/Oなどの選択されたデバイスがノーウェイトで応答するためには、このステートの間にDSACKx (Data Size ACKnowledge) 信号をプロセッサ側に返さなければなりません。

ステート3(S3) プロセッサは、このステートの最初でDSACKx信号が認識されないと次のステートに進まず、ウェイトします。

ステート4(S4) リードサイクルの場合、このステートの最後でプロセッサは入力データをラッチしますので、デバイス側はそれにまにあうようにデータバス上にデータを出力しなければなりません。

ステート5(S5) AS信号をネゲートします。デバイス側は、プロセッサがAS信号をネゲートするまではDSACKx信号をアサートしたままにしておき、AS信号がネゲートされてから1クロック以内にDSACKx信号をネゲートしなければなりません。アドレスやR/W信号は、このステートが終わるまで保持されます。

前にも説明しましたが、このDSACKx信号を認識してから1クロック後に取り込むというのが曲者で、遅すぎればウェイトが入ってしまいますし、かといって、DSACKx信号を返すのが早すぎれば、データがまにあわないうちに68030が取り込みを完了[*1]してしまいますから、ちょうどよいタイミングを作らなければなりません。

＊1
33MHzとか37MHzにクロックアップしたX68030でメモリアクセスが失敗するようになるのは、このためです。

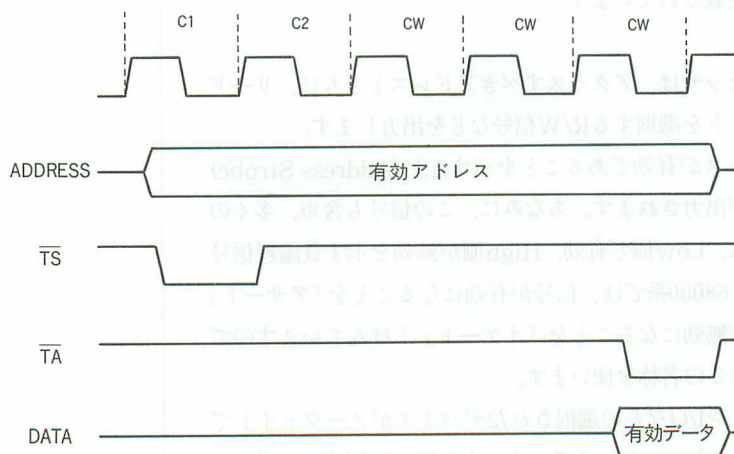



図2.2　68040のバスサイクル

●68040のバスサイクル

図2.2に3ウェイト*1で動作した場合のタイミングチャートを示します。68040では、クロックの立ち上がりエッジが各信号の基準になっています。

*1
68040の最小サイクルは2クロックなので、68030の2ウェイトは68040の3ウェイトに相当します。

クロック1(C1)　このクロックの前半で、プロセッサは、アドレスやR/W信号など、アクセスに必要な信号とともにTS（Transfer Start）信号をアサートします。
TS信号は次のクロックの前半にはネゲートされてしまいますので、デバイス側はクロックの立ち上がりエッジでTS信号をラッチして、アクセスの開始を認識しなければなりません。

クロック2(C2)　デバイスが応答できるときは、TA（Transfer Acknowledge）信号をアサートして応答します。プロセッサは、次のクロックの立ち上がりエッジでTA信号と、リードサイクルなら、データバス上のデータをラッチします。

ウェイト(Cw)　デバイス側の応答がまにあわない場合、TA信号を返さなければ、ウェイトクロックが挿入されます。

●変換回路

この変換回路の大ざっぱな信号の流れは、図2.3のようになります。基本は、68040のTS信号からX68030側に出力するAS信号を作り、X68030側のDSACKx信号から68040に返すTA信号を作るというわけです。

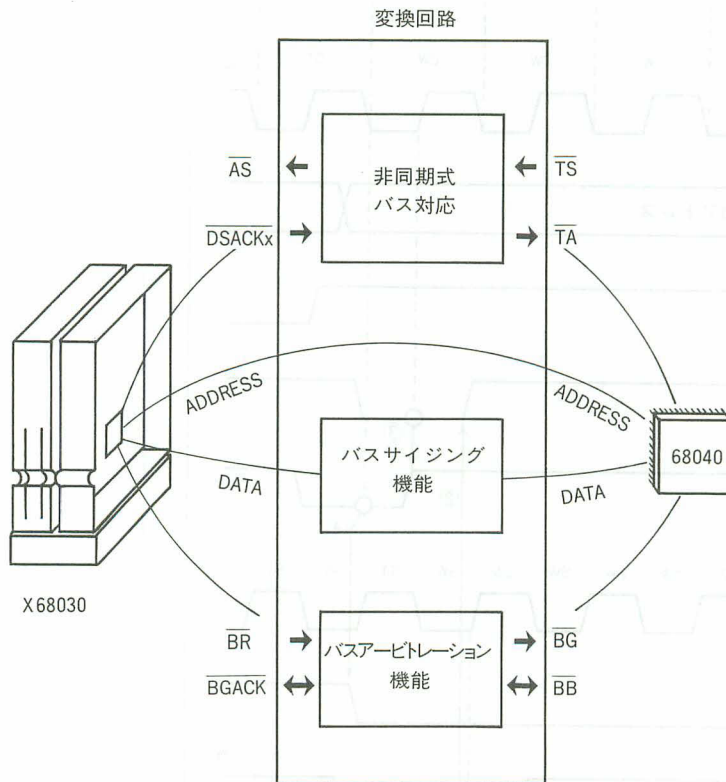



図2.3　変換回路の信号の流れ

この信号の流れにあわせて、68040とX68030側のバスサイクルのタイミングを書きなおしたのが、図2.4です。

68040のバスクロックに68030のクロックを反転した信号を使うことで、2つのバスサイクルのつじつまをあわせています。

①：TS　→ AS　アサート

②：DSACKx → Temp アサート（Tempは変換回路の内部信号）

③：Temp　→ TA　アサート

④：TA　→ AS　ネゲート

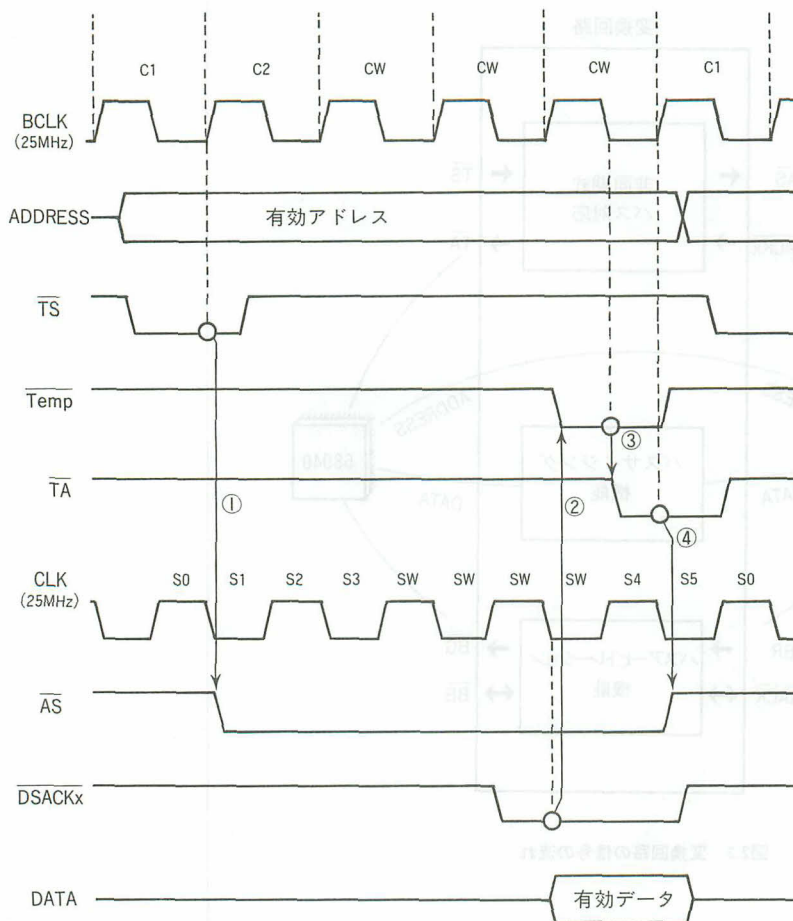



図2.4　変換した68040と68030のバスサイクル

C1/～S0　68040側からは、アドレスやR/W信号とともにTS信号がアサートされてきます。この後半が、68030側ではS0に相当します。

C2/S1～S2　このクロックの立ち上がりでTS信号をサンプリングし、68030側にAS信号として出力します。なお、TS信号は、次のクロックの立ち上がりまでにはネゲートされてしまうので、変換回路内でAS信号をループバックさせ、サイクルが完了するまでAS信号を保持しています。

Cw/S3～S4　C2の立ち上がりクロック、すなわち、68030側ではS3の最初で、DSACKx信号をサンプリングします。DSACKx信号がアサートされていれば、次のクロックでサイクルが完了するので、68040側へのTA信号がラッチされるように、S4の最初でTA信号のアサートを始めます。

C0/S5～S0　このクロックの立ち上がりで、68040側はバスサイクルを完了し、次のサイクルに入ります。変換回路も、このクロックの立ち上がりエッジではAS信号のループバックを解除してAS信号をネゲートします。

このタイミングで動作させれば、68040からX68030にアクセスできそうです。

ただし、これは、主要な信号*1について、マニュアルのタイミングチャートとつきあわせて考え出しただけで、細かい部分のタイミングは異なっています。

*1
TS、AS、DSACK 0、DSACK 1、TA。

とにかく、68040と68030が採用しているバスの転送方式自体が違っているのですから、完全に一致させるのは無理ですし、X68030側のメモリコントローラがどのようなタイミングで動いているか不明なので、タイミングマージンを見積もることができません。したがって、この検討は、

こんなもんなら動くかなあ。

程度のものでしかありません。

実際の微妙なタイミングの違いから、いろいろ問題が起こりましたが、この話は追って紹介していきます。

2）ダイナミックバスサイジングへの対応

前項ではデバイス側の応答をDSACKx信号として説明していましたが、実はこれはDSACK 0 とDSACK 1 という2本の信号線になっており、この信号の組み合わせで応答とともにデバイス側のデータバスのサイズをプロセッサに返すようになっています。

68030は、この応答を見てデバイス側のデータバスのサイズにあわせてバスを使い分けてくれるのです。この機能は、I/O系をX68000相当の16ビットバスで構成しているX68030には必須なのですが、68040にはないため、変換回路で16ビットサイズ[*1]へのバスサイジングをサポートしなければなりません。

この部分は、ちょっとやっかいなので、例を挙げて説明しましょう。

内蔵メモリとI/Oスロットに挿した拡張メモリとの比較で考えてみます。内蔵メモリは32ビットバスでつながっており、図2.5.aのような接続になります。これに対し、I/Oスロットは16ビットですから、拡張メモリのメモリマップは図2.5.bのような接続になります。このため、次の2つの問題がでてきます。

＊1
ロジックアナライザで見たかぎりでは、X68030では8ビットサイズの応答を返すデバイスはないようなので、8ビットサイズのバスサイジングはサポートしていません。

●ワードサイズのアクセス

ワードサイズでアクセスすることを考えてみましょう。内蔵メモリは32ビットデータバスですから、図2.5.aのように、0、4、8、C、…番地ではデータバスの上位16ビットを使って、2、6、A、E、…番地では下位16ビットを使ってアクセスします。I/Oスロットに挿した拡張メモリは、図2.5.bのようにデータバスの上位16ビットにしかつながっていません。68030は、ダイナミックバスサイジング機構によってデバイスのデータバスのサイズを判断して適切にアクセスします。

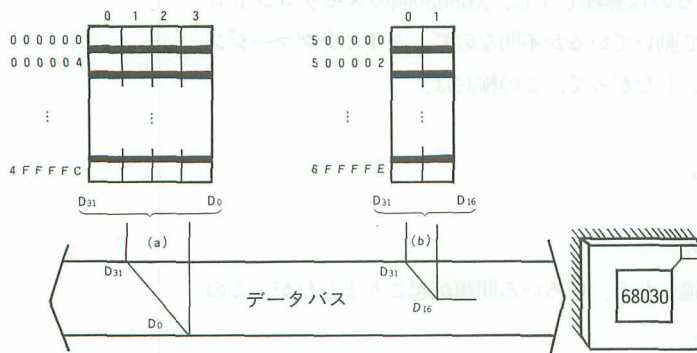



図2.5　データバスとメモリマッピング

しかし68040では、つねに32ビットバスのつもりで動作しますから、2、6、A、E、…番地のアクセスでは68040のデータバスの下位16ビットでアクセスしようとします。このため、図2.6のように変換回路でX68030側の上位16ビットと組み換えてやらなければなりません。

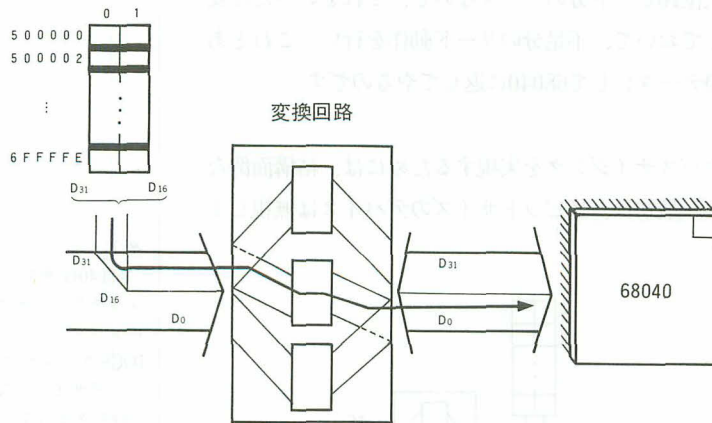



図2.6 バスの組み換え

●ロングワードサイズのアクセス

今度は、ロングワードサイズのアクセスの場合です。内蔵メモリは32ビットデータバスですから1回でアクセスが可能です。

I/Oスロットに挿した拡張メモリでは、16ビットしかバスがありませんから、一度にロングワードサイズのアクセスをすることは不可能です。ロングワードサイズのアクセスに対して、デバイス側がDSACKx信号により16ビットバスのアクノリッジを返してきた場合、68030はダイナミックバスサイジングにより、すぐに次のアドレスに対してワードアクセス*1を実行して不足データをアクセスし、あわせてロングワードとして扱ってくれるわけです。

しかし、68040はつねに32ビットバスのつもり*2でいますから、ロングワードサイズのアクセスができているように見せかけなければなりません。

このため、変換回路でダイナミックバスサイジングと同じようなことを実現してやります。ワードサイズのアクノリッジが返ってきた場合には、68040にアクノリッジを返さずにおいて、こっそり変換回路で次のワードのアクセスをしてロングワードがそろったところで68040にアクノリッジを返すという細工をするのです。ライトとリードで次のように処理しています。

＊1
X68000でロングワードのデータをアクセスするのと同じことです。

＊2
68040でも、2、6、A、Eといった番地から32ビット境界にまたがるロングワードデータをアクセスする場合は2回のワードデータアクセスとして実行します。

ライト　図2.7.aがライト時の動作です。DSACKx信号が返っても16ビットのデータバスだった場合は、68040側にTA信号を返さず、アドレスA1を立てて不足分のライト動作を行います。

リード　図2.7.bがリード時の動作です。こちらは、ちょっと厄介です。返ってきたデータは上位16ビット分のデータなので、これをいったん変換回路内で保持しておいて、不足分のリード動作を行い、これとあわせて32ビットのデータにして68040に返してやるのです。

このように、ダイナミックバスサイジングを実現するためには、結構面倒な処理をしなければなりません。最初は、16ビットサイズのデバイスは無視しようか*1とさえ思ったほどです。

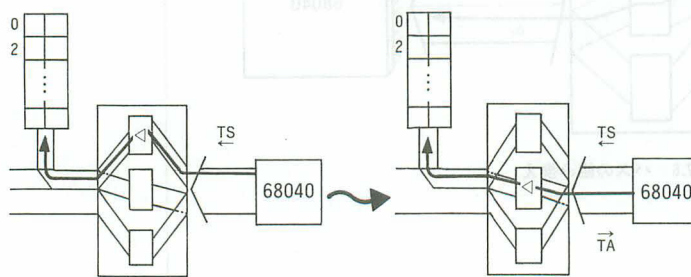
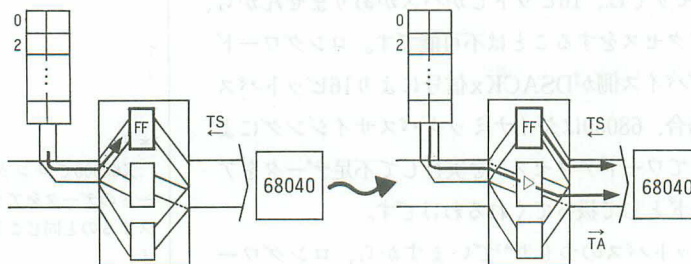
＊1
68040は通常のメインメモリアクセスだけのプログラムを実行し、IOCSコールやスーパーバイザモードなど、I/Oアクセスをしそうなときは68030に切り替えて実行するというアイデアです。プログラムが面倒だし、パフォーマンスが落ちそうなのでやめましたが、デバッグに苦しんでいたときに、何度もこの方法で逃げようと思いました。

(a) ライト時

(b) リード時

図2.7　ロングワードサイズのアクセス

３）バスアービトレーション機能への対応

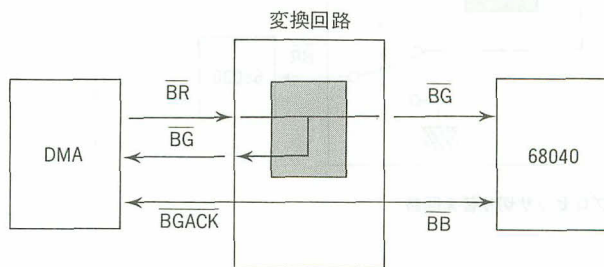
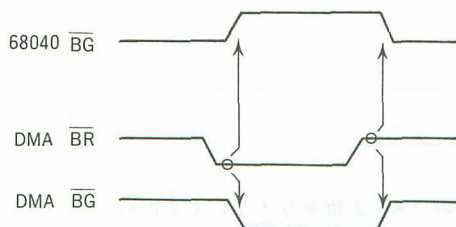
バスアービトレーションとは、プロセッサも含め、DMAなどバスアクセスを自発的に行うことのできるデバイスの間でバスの制御を調停する処理です。68030は、この調停処理を自分自身でしていました。DMAなどのデバイスは、BR（BUS Request）信号で68030にお伺いをたて、68030はBG（BUS Grant）信号でこれに答えるというわけです。

これに対し、68040は、DMAなどと同じように外部の調停回路[*1]にBR信号でお伺いをたてるようになっています。

このため、68030と68040ではBR信号とBG信号の意味がまるっきり逆になっています。変換回路では、簡単なバスアービトレーション機能によって、68040を含めたバスの調停を行っています。図2.8.aが、その大ざっぱな構成です。

図2.8.bが、そのタイミングです。基本的には68040のBG信号をアサートし、つねに68040がバスを使えるようにしておいて、外部デバイスからのBR信号を受けたときのみ、68040のBG信号をネゲートしてバスの制御権を手放させる[*2]のです。

＊1
マルチプロセッサ構成の場合、プロセッサ自身が唯一の調停者である68030は都合が悪いということで、こうなったそうです。

＊2
実際に68040がバスの制御権を手放したかどうかについては、変換回路では特に見ていませんが、68040のBB（Bus Busy）信号と68030側のBGACK（Bus Grant ACKnowledge）信号をつないでいるので、68040とバスを要求したデバイスとの間で、この信号を用いてうまく処理されます。

（a）アービトレーション構成

（b）アービトレーションのタイミング

図2.8　バスアービトレーション機能

このバスアービトレーション機能を拡張して、68030と68040の切り替え回路も盛り込んだのが図2.9です。スイッチによって、68030モードと68040モードを切り替えることができます。

68030モード　68040のBG信号を強制的にネゲートすることで、68040はバス制御権を得ることができず、事実上ストップ状態になります。

68040モード　68030のBR信号を強制的にアサートすることで、68030はバス制御権を外部デバイスに渡したまま返してもらえず、事実上ストップ状態になります。

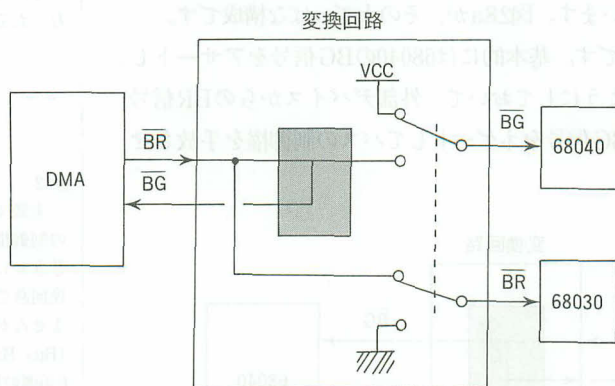



図2.9　プロセッサ切り替え回路

COLUMN

68030と68040のマルチプロセッサ化?

2つのプロセッサをバスにつなぎ、片方のバス制御権を与えないまま止めておくやり方は「68020 on X68000」のときに実績がありましたので、今回もこの方式にしました。

せっかく68030がバスアービトレーション機能を持っているのだから、68030を生かした状態にし、68040が68030に対してバス要求するという構成も考えていたのですが、複雑になりそうなので見送りました。

ハードウェア工作

ゴールデンウィークの間*1で、なんとか大体の回路の目安がつきました。しかし、68030と完全に一致する信号を作ったわけではありませんし、そうする意味もありません。実際にはX68030側が使っていない信号は切り捨てて*2シンプルにするほうが得策です。X68030の回路図やメモリコントローラなどのカスタムチップの仕様がわかれば、設計段階で詳細に煮詰めていくことができますが、残念ながら、こういったハードウェアの資料は公開されていません。

また、回路も目安がついただけで、詳細部分は実際に動かしてみないとよくわかりませんから、まずはバラック基板で実際に回路を組み立てることにしました。

とにかくこれは動くことが目的なので、必要最小限のシンプルな回路としました。２次キャッシュやローカルメモリを持つと、68030バスに変換する際のオーバーヘッドが低減するので高速化が期待できますが、バラック基板で作るのは容易ではありません。また、機能を増やせば、それだけハードウェアのデバッグも大変になります。動かなければせっかく作った基板もゴミと化しますので、机上ではあれこれ夢を広げましたが、バラック基板で組み立てようと決意したとき、きれいさっぱりあきらめました。

それでもまったく速くならないのでは、前に作った「68020 on X68000」と同じ運命をたどることになってしまいます。ざっと試算してみると、68040のパフォーマンス自体が高いので、なんとかX68030の２倍程度にはなるだろうとの感触を得て、１年ぶりに本格的な電子回路工作*3に没頭する日々が始まりました。

GALライタを作る

個人の電子回路工作といったら、普通は標準ロジックICの74シリーズ*4と相場は決まっているのですが、74シリーズは１つのチップにANDが４個、別のチップにORが４個といった感じで単機能ごとになっており、今回のような変則的な回路を構成しようとすると多数のチップを組み合わせなければならず面倒です。

こんな用途に便利なのが、「PLD」(Programmable Logic Device) と呼ばれる、プログラム可能なチップです。PLDのなかでも有名なのが、PAL (Programmable Array Logic) と呼ばれるシリーズで、前に作った「68020

＊1
昼間は家族サービスもあって時間がとれないので、主に夜ですが。

＊2
実際、ECS、OCS、RMCなど、最初は多くの信号を変換回路で作っていましたが、X68030側が使っていないようなので、切り捨てました。

＊3
この間もコネクタを作ったり、裸のハードディスクを増設したり、ということはしていました。

＊4
TTL（Transistor Transistor Logic）の代名詞として使われているくらい有名なICです。

on X68000」でもPALを使用しました。このとき使ったPALは1回プログラムを書き込んだらあとは書き換えできないものでしたが、もともと、この回路は本に出ていた回路を参考にしていたので、基本的には動くだろう*1と、いきなりパーツ屋で焼いてもらいました。

しかし、今回は変換回路の目安がついただけで、自分でいうのもなんですが、動く可能性はゼロでしょう。試行錯誤でどんどん回路を変更していくことになりそうですから、1回しかプログラムできないPALではつらかったのです。

そこで、今回はGAL(Generic Array Logic)*2を使うことにしました。GALの場合は再プログラム可能です。

また、手元で焼けるようにしたかったこともあって、前からチェックしていた資料*3をもとに、まず、GALライタ*4を作りはじめました。写真2.1が自作したGALライタです。このGALライタはセントロニクスのプリンタポートに接続して使うようになっており、書き込みタイミングなどもすべてソフトウェア制御なので、ハードウェアは驚くほどシンプルにできています。

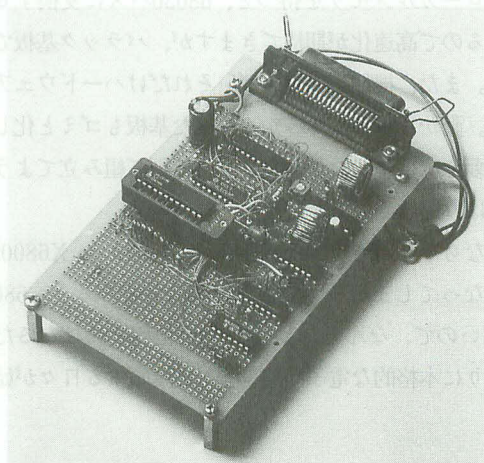



写真2.1　自作したGALライタ

このGALライタでは、16V8と20V8という、2品種のGALのそれぞれノーマルバージョンおよび高速化したAバージョンを焼くことができます。後に、より高速なBバージョンを使うために、結局、市販のGALライタ*5を購入することになりましたので、今では使っていませんが、バラック基板の立ち上げのときは活躍してくれました。

動くかどうかもわからないもののために、いきなり市販のGALライタを買う勇気はなかったので、もしこのGALライタがうまく動かなかったら、変換回路の製作は投げ出していたかもしれません。

＊1
それでも、2回間違えました。

＊2
PALよりも高機能で、PALの多くのシリーズをGALで置き換えることが可能ということで、Genericという名前になっているのです。

＊3
『簡単にできるGALライタの製作』トランジスタ技術誌1991.7

＊4
これを作るのに、結局、半月くらい費やしました。

＊5
GALトースターKEM-907GAL。

バラック基板

バラック基板で回路を作るといっても実際には簡単ではありません。配線量も膨大です。しかし、実は最も面倒なのが、マザーボードのプロセッサのソケットから信号を取り出すための接続ピンを作る作業なのです。「68020 on X68000」では、写真2.2のように、68000のシュリンクピッチの足にあうように基板にピンを植え、その上にICソケットを載せて半田付けをしました。ここに68000を載せるとともに、信号を取り出したわけです。

しかし、68030はPGA（Pin Grid Array）と呼ばれる、剣山*1をひっくり返したようなパッケージなので、同じことをやろうとすると、格子状に百数十本のピンを植えなければなりません。また、PGAのソケットにピンをいきなり半田付けすると、断線したとき、PGAのソケットの内側のほうは直せなくなる恐れがあります。このため、今回は図2.10のように、２階建ての基板とし、一方の基板にマザーボード側につなぐ接続ピンを、他方の基板にICソケットをつけ、その間を別のコネクタで接続しました。

＊１
お花を生ける際に使う「けんざん」と呼ばれる金属製の針を多数植えたもの。うっかり剣山状態になったPGAを踏んづけた人の話とか、よく聞きます。

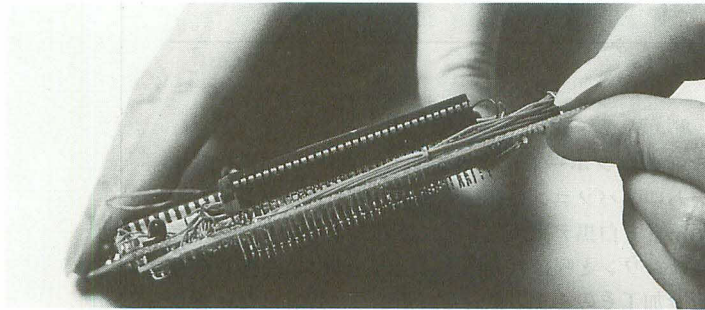



写真2.2　X68000本体との接続部分

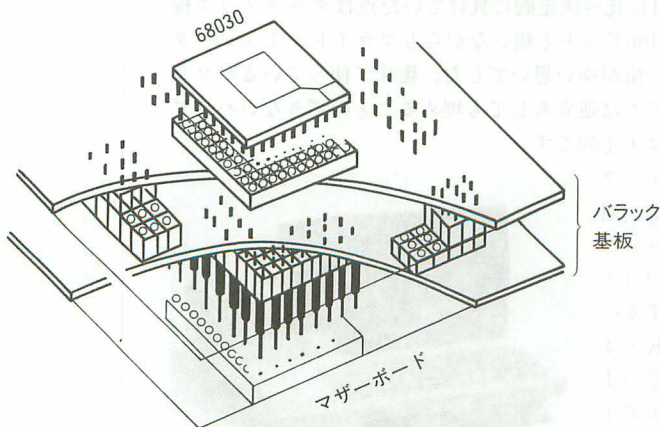



図2.10　X68030本体との接続部分

次ページの写真2.3.aがこのバラック基板の外観です。写真2.3.bが横から見たところで、メインの２階建ての基板の１階部分にマザーボードとの接続ポートとダイナミックバスサイジングのための回路が入り、２階部分には68040と68030が、そして、建て増しした３階部分の基板にGALによって構成される制御回路が載っています。

68030や68040はピン数が多いので、ピンから電線を引き出すと、配線が交錯してきてわけがわからなくなります。これを回避するため、メインの２階建て部分の基板は、68030のテストポッドでやったように、ピンの足からプリントパターンで信号線を引き出すようにしました。

COLUMN

名機MZ-80K

MZ-80Kは、TK-80などのワンボードマイコンから、キーボードとディスプレイを装備した、いわゆるパソコンに移行する頃に生まれたマシンです。おもちゃみたいなキーボードと白黒ディスプレイ、それにカセットテープデッキを一体化したオールインワンと呼ばれる構成は、PC-8001の洗練されたデザインと比べると、板金加工そのままの無骨なデザインは野暮ったい感じでしたが、ワンボードマイコンの雰囲気を色濃く残していて好きでした。

当時、MZ-80KがPC-8001に比べ決定的に負けていた点はグラフィック機能で、PC-8001上で、160×100ドットと粗いながらもフライトシミュレータなどが走っているのを見て、歯がゆい思いでした。趣味で使っているパソコンだからこそ、ソフトウェアでは逆立ちしても埋めることのできないハードウェアの機能の差というのは決定的です。

そこで、なんとかグラフィックをやりたいと考え出したのが、MZ-80K用のグラフィックボードです。320×200の解像度（8Kバイト）のフレームバッファを実現するのに、2114というSRAM（１K×４ビット容量）を16個も積んでいました。これが私のハードウェア工作の原点になっています。

このプリントパターンは、サンハヤトという、電子回路の工作をやったことがある人にはおなじみのプリント基板の手作りキットのシリーズの、フォトエッチングタイプを使いました。ちなみに、私が最初に基板を手作りしたのはMZ-80Kのグラフィックボードでした。

このグラフィックボード作りでは、ずいぶんいろいろな経験を重ねました。フォトエッチングは、透明なシートにパターンを描き、蛍光灯で焼き付け、現像液に浸して現像し、エッチング液につけて余分な銅箔を融かすのですが、これがなかなか大変です。現像時間をミスると、焼き付けたパターンが流れてしまったり、逆に不要な部分が残ったりしますし、エッチング時間をミスると、パターンが溶けて断線したり、なかなか溶けずにショートしたりします。結局、MZ-80Kのグラフィックボードだけでも４枚は作りましたので、最後のものはかなり高い完成度を誇っていました。

ただし、今回の回路は規模が段違いに大きく、すべての配線をプリントパターン化することは到底無理です。ソケットの部分と基板の１階と２階をつなぐ部分、それに回路的に大体フィックスしている、バスサイジングのためのバス組み換え部分に使っているだけで、後のほとんどの配線は電線を１本１本張っていきました。

あえてフォトエッチングでプリント基板を作ったのは、久しぶりにプリント基板を手作りしてみたかったのと、スルピンキットという、お手軽にスルーホール*1を作るツールを買ったので、これを使ってみたかったという趣味的要因が大きいといえます。

＊1
普通は、基板に穴を開けた後、穴の内壁を金属メッキします。こうしておけば半田付けするときの接触面が大きくなるというメリットのほか、両面基板の表と裏のパターンをつなぐ意味もあります。スルピンキットは後者の目的で使いました。

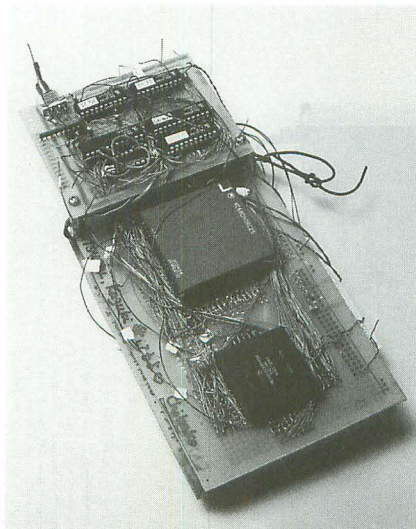



写真2.3　040turboバラック基板　a.外観　　b.側面から見たところ

まずは68030

いろいろ苦労はしましたが、なんとかバラック基板のパターンができあがりました。この時点では、まだ68040や変換回路はついておらず、68030が2階建ての基板を通してマザーボードにそのまま接続されているだけです。この状態で、まず、68030が動くかどうか確認します。

X68030のマザーボードにつながるピンと、バラック基板上の68030のICソケットとの導通を確認してから、いざ取り付けというときになって、はじめて基板がX68030の前面パネルに当たってしまうことに気づきました。68030の真上と後ろのビデオユニットとの間に余裕がないことには注意していたのですが、前にもはみ出してしまうのです。しようがないので、右タワーの前面パネルも取り外すことにします。

ちなみに、この前面パネルを固定するネジは、X68030の左右のタワーの隙間にあるので、普通のドライバーでは回せません。しかたがないので小振りプラスドライバーの先を曲げ、写真2.4のようにして、これを使ってやっと外すことができました。写真2.5がバラック基板を取り付けたところです。

写真2.4　前面パネル用ドライバー

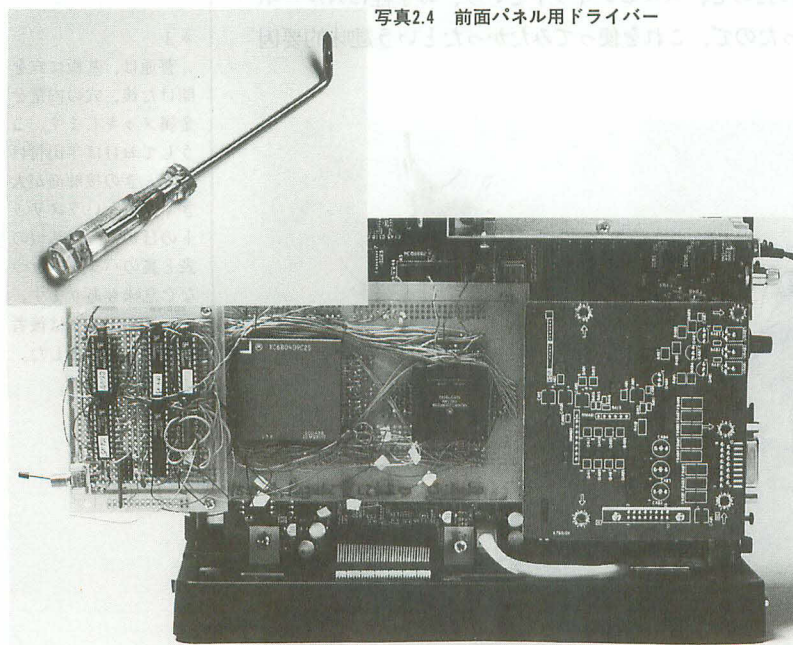



写真2.5　バラック基板を取り付けたところ

やっとの思いで取り付けて、接触不良などの問題も克服して、なんとかうまく68030を動作させることができました。単に信号線を引き出しただけなので、これは動いて当然と思われるかもしれませんが、後で思うと、ここで問題が出なかったのはラッキーでした。

というのも、040turboの基板第一次配布の結果、68030モードが正常に動作しないという不具合が２件発生したのです。どうもX68030には個体差があるようで、040turboを取り付けると、信号線が伸びたせいか、あるいは負荷が増えたためか、040turbo上の68030でデータを取りこぼすようになるのです。これは、マザーボードに手を加えることで事無きを得ましたが、そもそも040turboが動くという確信があったからこそ、マザーボードの信号の状態を追究することができたのです。我が家のX68030がこの手のマシンだったら、プロセッサソケットからの信号線取り出しは無理とあきらめ、この時点で040turbo計画は消えていたでしょう。

次は68040

次は、マザーボードから引き出してきた68030の信号に68040の対応する信号をつなぐ作業です。

「68020 on X68000」の基板では、68000と68020の対応する信号をつなぐのに、プロセッサのピンに直接リード線を半田付けしていきましたので、基板の裏面の68020のピンに配線が集中してしまい苦労しました。今回はその教訓を生かして、プリントパターンでプロセッサのピンから信号線を引き出しているので、基板の表面で配線するだけですみます。

もっとも、そのためにパターンを描いてプリント板をエッチングし、穴を開けてスルピンキット*[1]でスルーホールを作るという作業が必要になりました。損得勘定からすると、これで得だったかどうか*[2]は、わかりません。

ともかく68040と68030の間の配線をします。68030部分には手をつけずにそのままにしておき、68040のアドレスバスはそのまま直に68030側のアドレスバスに、データバスはバスサイジング回路を経由して68030側のデータバスにつないでいきます。

この時点では変換回路はまだできていませんので、68040のBG信号とバスサイジング回路のゲート信号は、とりあえず５Ｖにつないで動かないようにしておきます。BG信号が有効になりませんから、68040はバス制御権をもらえ

＊1
とても簡単にスルーホールができるので、最初は嬉々としてやっていたのですが、さすがに数百ヵ所もやると嫌になってきます。

＊2
前にもいったように、プリント基板を作りたくて始めたのですから、自業自得なのですが。

ず、リセットが解除されてもバスにつながっているだけで、事実上ストップした状態になります。これは、68030と68040の切り替え回路の作り方のところで説明した方法と基本的に同じなので、この状態で68030が正常に動作しなければ、プロセッサの切り替え回路自体を考えなおさなければなりません。

とりあえず、ここまで作った状態で、再度、接続テストをしてみます。

単に68040がバスにぶら下がっているだけですから、動いて当然なのですが、素直に動いてはくれませんでした。もともと基板の仕上がりもよくなかったこともあって、作業途中で断線が多数発生しました。特にコネクタが多かったので、X68030のマザーボードへの取り付け・取り外しの際や、配線のために2階建てになっている基板をバラしたりするときに、物理的なストレスがコネクタ部にかかってコネクタと基板のパターンの間が必ずどこか断線しました。

最初の頃は、回路の修正をしているのか、基板を壊しているのかわからないという情けない状況で、1回の取り外し作業ごとに200ヵ所以上あるコネクタ関連の導通をチェックしなおさなければなりませんでした。

そうこうして、なんとか68030を動作させることができました。

いよいよ変換回路

さて、いよいよ変換回路の製作に取りかかります。

今までは、マザーボードの68030ソケットの信号を、そのままバラック基板上の68030の信号とつないでいただけですが、一部の信号を切断*1して、変換回路のプロセッサ切り替え回路経由で接続することになります。本当は、この変換回路もバラック基板の2階部分に作るつもりでしたが、穴開け作業をすると*2、またパターンが切れそうだったので、ユニバーサル基板を建て増ししました。見栄えが悪いのはしかたがありません。なんとか作業が終わってGALを載せ、最初の回路ができあがりました。しかし、さすがにこれはまったく動きませんでした。

X68000からそうですが、X68030の前面のパワースイッチは、オンにする場合のみ機械的に電源が入るようになっており、オフにする場合はソフトウェア割り込みになるだけです。実際の電源切断はプログラムで行うようになっています。したがって、プロセッサが正常に動かないと電源オフさえできなくなるので、背面のメインスイッチで何度、強制的に電源切断したかわかりません。

68030さえも全然動かないのでは話にならないので、まずはGALを全部外

＊1
もちろんバラック基板上での話です。

＊2
最初にやっておけばよかったのですが、面倒だったのでサボっていました。

して、空いたGALソケットに、68030モードのときにつながるであろう信号どうしを、ジャンパー線を挿し込んで接続してみます。そうすると、とりあえず68030は動くようになりました。こうなると、問題があるのはGALのロジックということになります。今回は、そもそもの設計ミスなどもありましたが、GALを使うのもはじめてなら、GALのプログラミングに使用したCUPL*1というコンパイラに接するのもはじめてだったので、GALの使い方のミスや勘違いもありました。

この際、１つ失敗例を紹介しておきましょう。

*1
GALやPALの書き込みデータは、結線のオン・オフの羅列で、とても手作業で作れるものではありません。このため、普通は、ANDやORといった論理式のレベルでソースファイルを作り、専用プログラムを通して書き込みデータを作るのです。

GALのプログラム

GALのプログラムでは、出力ピンに使用できる機能は、入力ピンの各信号にANDやORなどの単純な論理式を通して得られる結果に加え、フリップフロップ出力やトライステート出力を選択することができます。

フリップフロップ出力は、図2.11.aのように、トリガ信号が立ち上がるとき(立ち下がりで働くものもある）の入力信号の状態を取り込んで出力するものです。以後、入力信号が変化しても、次のトリガが入るまで同じ出力を保ちます。

トライステート出力は、通常のTTLロジックで使われるHigh（2.4V以上）とLow（0.4V以下）の２つの状態のほかに出力をカットするという第３の状態を持つものです。これは、回路記号では図2.11.bのように書かれ、ゲート信号で、この第３の状態を制御します。ゲート信号がオンのときは、入力信号の状態が出力信号に伝えられますが、ゲート信号がオフになると、出力信号は回路から切り離された状態になります。この状態を「ハイインピーダンス」と呼びます。バスのように１本の信号線上に多数の出力回路がつながる場合は、出力しない回路をハイインピーダンス状態にすることで、出力が衝突しないようにします。

図2.11　フリップフロップとトライステート

さて、今回使った16V8および20V8というGALは8本の出力ピンを持っており、1本ごとに組み合わせ出力とフリップフロップ出力を指定できます。ここで、組み合わせ出力の場合は、トライステート出力にした場合のゲート信号に任意の入力ピンを使用することができますが、フリップフロップ出力にした場合は、トリガ信号およびトライステートのゲート信号に使える入力ピンが決まっているのです。

CUPLの表記法でいくと、図2.12.aのような回路は、次のように書けます。

```
Pin 2 = A;
Pin 3 = B;
Pin 19 = OUTPUT;
OUTPUT = A AND B;
```

出力をトライステートにした図2.12.bの場合は、次のように書きます。

```
Pin 18 = C;
OUTPUT = A AND B;
OUTPUT.OE = C;
```

それでは、これをフリップフロップ出力にした場合はどうでしょう。".D"をつけるとフリップフロッピ出力になりますので、

```
OUTPUT.D = A AND B;
OUTPUT.OE = C;
```

と書きたくなりますが、実際にできる回路はこうなりません。入力信号Cでは出力OUTPUTのトライステート制御ができず、16V8の場合なら、図2.12.cのように、11番ピンに固定*1です。CUPLがエラーにしてくれればまだいいのですが、これが通ってしまうので、間違いに気づきにくいのです。

最初からフリップフロップで設計しているところはいいのですが、組み合わせ出力のつもりでいて、後でフリップフロップ出力に変更した場合、トライステート制御がCの入力信号で制御できなくなっていることに気がつきません。エラーにならないのでOKと思っていたのですが、実際にはトライステートになっていなかったというわけで、期待したものとは違う回路になってしまうわけです。

このため、68030モードではハイインピーダンスになっていなければならない信号がハイインピーダンスにならず、変な信号を出して、68030の動作を妨

*1
フリップフロップ出力でなければ、トライステート制御を自由に割り付けられます。

害していたのです。

なんでGALからAS信号が出っぱなしになってるんだ？

手作りGALライタを使っていたので、うまくGALが焼けていないんじゃないかとか、そもそもGALが不良品なんじゃないかとか、理由がわからず苦労しました。

また、このことに気づいてからも、何回もついうっかり間違えてしまいました。

CUPLがエラーにしてくれればいいのに

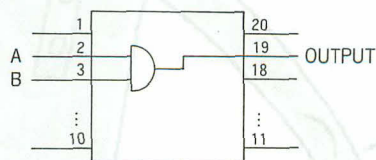
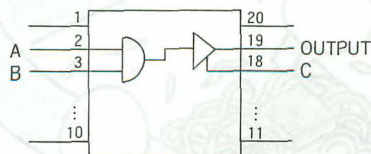
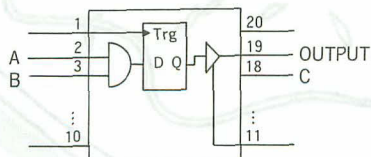
と、自分のミスを棚に上げて、八つ当たりしたものです。

（a）組み合わせ出力

（b）トライステート出力

（c）フリップフロップ、トライステート出力

図2.12　CUPLの記述から生成されるGALの回路（16V8の場合）

ここここは接続を確認したけど、
アレッ、ここはどうだったかな？

第3章

68040の胎動

68040モード

回路の本格的なデバッグに入る前に、いろいろ問題が出ましたが、はじめての経験が多かったので、これはしかたがないでしょう。この段階までは、ハードウェア工作のレベルだったので比較的楽な作業でしたが、いよいよこれから、しんどくなります。

GALの論理を修正して、なんとか68030モードが動作するようになりました。パワースイッチをオフにすれば電源も落ちるようになりました。

よし、次は、68040モードだ。

切り替えスイッチを68040モードにしてパワースイッチをオン。

やはり、というより、期待もしていませんでしたが、まったく動きません。パワースイッチをオフにしても、当然、電源は落ちません。ただ、68030モードに戻してリセットをかければ、68030がパワースイッチのオフを認識して電源を落としてくれるので、背面のメインスイッチのお世話にはならなくてすみます。

さっそく、ロジックアナライザをGALの変換回路につないで電源をオンにして各信号線を調べてみます。このときの68040の信号線は次ページの図3.1のような状況になっていました。

1. リセットが解除される。
2. 68040がTSをアサートする。
3. 変換回路がこれを受けて68030側にASをアサートする。
4. しばらくすると、68030側からDSACKxが返ってくる。
5. 変換回路がこれを受けて68040側にTAを返す。
6. 68040がTSをアサートする。
7. 変換回路がこれを受けて68030側にASをアサートする。
8. しばらくすると、68030側からDSACKxが返ってくる。
9. 変換回路がこれを受けて68040側にTAを返す。

これで終わりでした。リセットボタンを押すと、2回だけアクセスが起こります。この2回のアクセスは、たぶん、イニシャルスタックポインタの読み出

しとリセットベクタの読み出しをしようとしているのでしょう。

たったこれだけの動作をしただけで後はダンマリになるのですから、暴走というより、まったく動いていないといったほうがよい状況です。しかし、これを見ていて、なんだかうれしくなりました。駄目でもともと、動く自信なんてぜんぜんなかったバラック基板でしたが、68040がなんとか動こうとしてくれているのです。変換回路もけなげに動いているようです。大袈裟かもしれませんが、それは、68040の胎動のように思えたのでした。

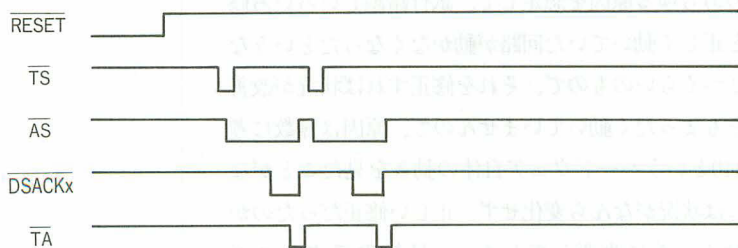
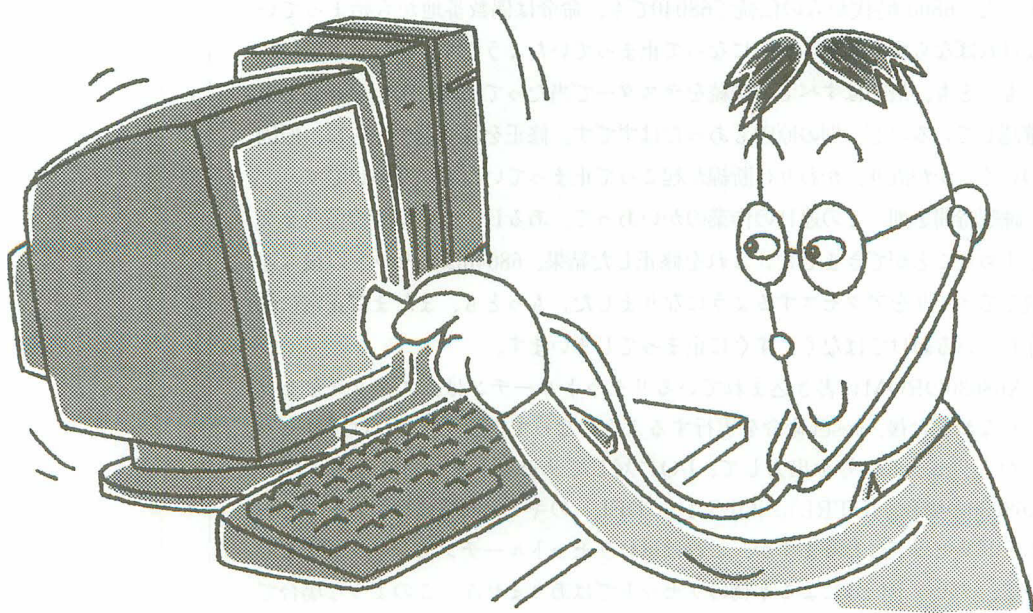



図3.1 最初のアクセス

「ドックン、ドックン」これが68040の心音なのかな？

確かな手応え

それからは、なぜ2回のアクセスで止まってしまったのかについて、あらゆる箇所を調べました。ほとんど動かないという状況ですから、動かない原因を絞り込めるほど情報があるわけではありません。

およそ、思いつくかぎりのあらゆる原因を想定して、試行錯誤でいろいろ修正してみましたが、もともと正しく動いていた回路が動かなくなったというなら、動かない原因は1つか2つくらいのもので、それを修正すれば状況が改善されますが、今回は、そもそもまったく動いていませんので、原因は無数に考えられます。そのうえ、68040というハードウェア自体の動きを見たことがないので、修正しても見た目には状況がなんら変化せず、正しい修正だったのかどうかさえ判断がつきません。逆に改悪してしまい、最初のアクセスでDSACKxが返ってこなくなったりしたこともありました。

動かない原因はいろいろあったのでしょうが、最終的にはデータバスの断線が原因だったようです。2階建ての1階にバスサイジングの回路があり、2階に68040が載っているわけですが、この間の接続に断線があって読み出したデータが化けたために、リセットベクタが奇数番地を指すような格好になっていました。68000時代からの伝統で68040でも、命令は偶数番地から始まっていなければならず、内部エラーになって止まっていたようです。

もっとも、最初はすべての接続をテスターで当たって断線していないことを確認しているので、別の原因もあったはずです。修正を繰り返すうちに、原因のいくつかが直り、かわりに断線が起こって止まっていたというわけです。

睡眠時間を削っての連日の作業のかいあって、ある日、ついに断線箇所を突き止めることができました。これを修正した結果、68040はリセット以後も連続してメモリをアクセスするようになりました。もっとも、まだまだ正常に動作しているわけではなく、すぐに止まってしまいます。

X68030のROMに書き込まれているリセットルーチンは、スタックをセットしなおした後、reset命令を実行するようになっています。これはプロセッサからリセット信号を出力して、I/Oに対してリセットをかける命令です。X68030の場合、[CTRL]+[OPT.1]+[DEL]のキーを同時に押すことでキーボードリセットになりますが、これは単にリセットルーチンにジャンプするだけで、リセット信号による本物のリセットではありません。このような場合でも、reset命令を実行することで、I/Oに対し、確実にリセットをかけること

ができるのです。

そして、68040ですが、リセット解除後、少しアクセスをした後、ちゃんとRSTO (Reset Out) 信号をアサートしてきました。よちよち歩きのようなものですが、けなげにもreset命令を理解しているのです。

這えば、立て。立てば、歩めのX68ゴコロ。

デバッグの日々　その1

68040が連続してメモリをアクセスするようになったといっても、プログラムがまともに動いているわけではありません。画面はあいかわらず真っ黒ですし、しばらくするとダンマリになってしまいます。reset命令を認識しているようですから、ROMからはプログラムをうまく読み込んでいると思っていいでしょう。

どこでコケてるのか？

これが、次の課題になります。

ハードウェアだけ見ていてもらちが明かないので、X68030のROM内のルーチン＄FF0000～＄FFFFFFを逆アセンブルして解析することにしました。できたリストはファイルにして約300Kバイト。2段組で縮小印刷をしても、優に300枚を超えるものになりました。

昔、X68000の未公開IOCSを解析するため、同じように大量の逆アセンブルリストと格闘したことがありましたが、あの場合はIOCSコールごとに動作が分かれていましたので、比較的簡単に処理を追うことができました。

しかし、今回は、延々と続くプログラムを解析しなければなりません。アセンブラは自分でプログラムしても、しっかりしたコメントをつけておかないと、何をやっているのか後でわからなくなります。ましてや逆アセンブルしただけのコメントなしのリストです。最初は何をやっているのか、さっぱりわかりませんでした。

とりあえずリストと照らし合わせながら、68040がどこまで走っているのかを調べることにします。しかし、これも一筋縄ではいきません。チャンネル数の多い、ちゃんとしたロジックアナライザならアドレスバスの変化を追うことで処理の流れがわかるのですが、私の使っているロジックアナライザは8チャンネルしかないので、アドレスバスの変化を一度に見ることができないのです。

しょうがないので、8チャンネルのうちの2チャンネルをTS信号とTA信号につなぎ、残りの6チャンネルをアドレス線の6本につないで、リセット後のアドレスの変化をメモしていきます。そして、アドレス線6本をつなぎかえて、またリセットからの変化をメモしていきます。この作業を4回やって、やっとアドレス線24本分の変化を突き止めました。

ちなみに、68030のアドレス線は32本ですが、X68030はこのうちの24本しか使っておらず、残りの上位８本はどこにもつながっていません。このため、たとえば、＄00123456にアクセスしても、＄FF123456にアクセスしても、同じメモリにアクセスしたことになります。このため、X68030でのアドレス空間は16Mバイトが限界になります。

さて、アドレスの変化を追っていくと、どうもサブルーチンから戻ってくるときに暴走しているらしいことがわかりました。サブルーチンは、戻り番地をスタックに書き込みます。書き込みに失敗して変な戻り番地をセットしたために、リターンに失敗したか、はたまたリターン時の読み込みに失敗して変な番地に戻ろうとしているのかはよくわかりません。これを確認するためには、今度はデータバス32ビットの監視が必要になりますが、さすがにこれは無理です。

原因の詳細はわかりませんが、現象が絞れてきました。注目すべきことは、このスタック操作ではじめてRAMアクセスが行われているということです。ROMアクセスはうまくいっても、RAMアクセスに失敗している可能性が高いのです。

結局これは、変換回路の作るタイミングが68030のものと微妙に違うために、スタティックカラムモードのシビアなタイミングに追従できていないのが原因のようでした。スタティックカラムモードは、写真3.1のように、マザーボードのメモリコントローラの近くにある、SW１と描かれたパターンをジャンパー線でショートさせることで禁止することができます。ためしにこれをやってみたところ、RAMアクセスが正常に行われるようになりました。

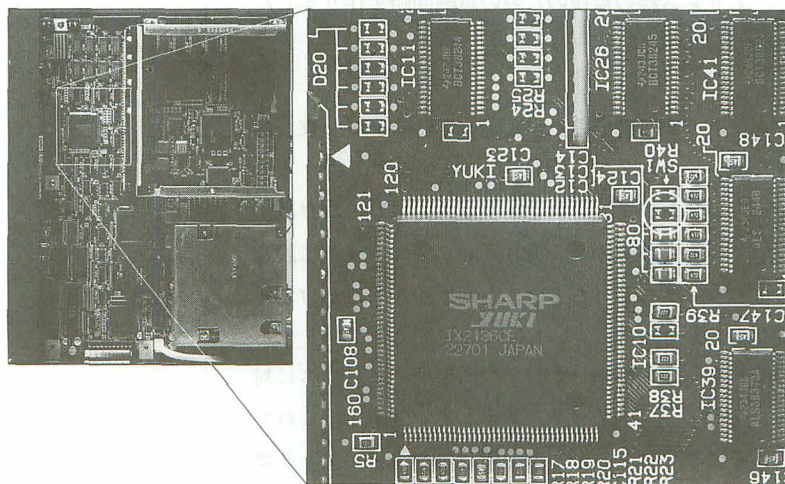



写真3.1　スタティックカラムモードをオフにする設定

このスタティックカラムモードを禁止するSW1の情報は、X68030のクロックアップに関する情報といっしょにパソコン通信上で知ったものです。当時、よく、こんな設定を見つけたものだと感心したのを覚えています。もし、この設定を知らなかったら、これまた040turboはここで消えていたかもしれません。

なお、後にスタティックカラムモードも使えるようになりました。

デバッグの日々　その2

スタティックカラムモードを禁止することで、また少し進むようになりましたが、まだまだ動いているというには程遠く、しばらくするとダンマリになってしまいます。

X68030のROMルーチンを解析して少しわかったことは、68000から68030までの各プロセッサについては、プログラムが自分で判定[*1]するようになっているということでした。しかし、68040を判定するルーチンは入っていません。もしかしたら、このROMで68040を動かすのは無理なのかもしれないという考えが頭をよぎります。いちおうのアクセスができているのだから、68030でHuman68kを立ち上げた後、動作プロセッサを68040に移行するようにしたほうが早くできるかもしれないと思いはじめました。

しかし、具体的に何が68040の動作を阻んでいるのかが気になるところです。

さすがにアドレスの変化を8チャンネルのロジックアナライザで監視するのはあまりに面倒だったので、アドレス線の監視のための試験回路を作りました。次ページの図3.2が、そのブロック図です。

＊1
68020で追加された命令を68000で実行すればエラーになったりしますから、こうやっていくつかの命令を実行してみれば、切り分けがつくのです。

比較部は、74ALS688というチップで構成され、A端子とB端子が一致したときにEQ端子がアサートされます。EQ出力にロジックアナライザの1チャンネルをつないでおいて、これをトリガ入力にしておけば、A端子側につないだアドレス線がB端子側のディップスイッチに設定した値と同じビットパターンになったとき、トリガがかかるわけです。前後のアドレス変化はわかりませんが、ROMの逆アセンブルリストをにらんで処理の流れを一歩一歩追いかけて要所要所のアドレスに対するビットパターンをディップスイッチに設定していきます。これでトリガがかかれば、そこまでは走っていると確認できますし、トリガがかからなければ、その前のどこかでコケたというわけです。ソフトウェアのデバッグで使うブレークポイントのイメージに近いといえる

でしょう。

表示部は、74ALS273というチップで構成され、CK端子の信号の立ち上がりでIN端子につないだデータを取り込み、OUT端子に出力します。CK端子に68040のTS信号をつないでおくと、プロセッサがアクセスしようとするアドレスが次々と取り込まれます。68040が正常に動いているときはアドレスを読み取れませんが、68040が黙り込むと、最後にアクセスしようとしたアドレスがわかります。また、アクセス自体は生きていて無限ループに陥っているような場合は、適当にMSK端子のスイッチを押して一時的にアドレスの取り込みをやめれば、どのあたりでループしているかがわかります。

簡単な回路でしたが、これで、だいぶデバッグが楽になりました。

話を本筋に戻しましょう。ROMのどのあたりでおかしくなるかを調べていくと、どうも＄FF0000番地より前に飛んでいって暴走しているらしいということがわかりました。何をやろうとしているのでしょう。ここで、ふと、もしかしたらROMデバッガがいけないのかもしれないということに気づきました。

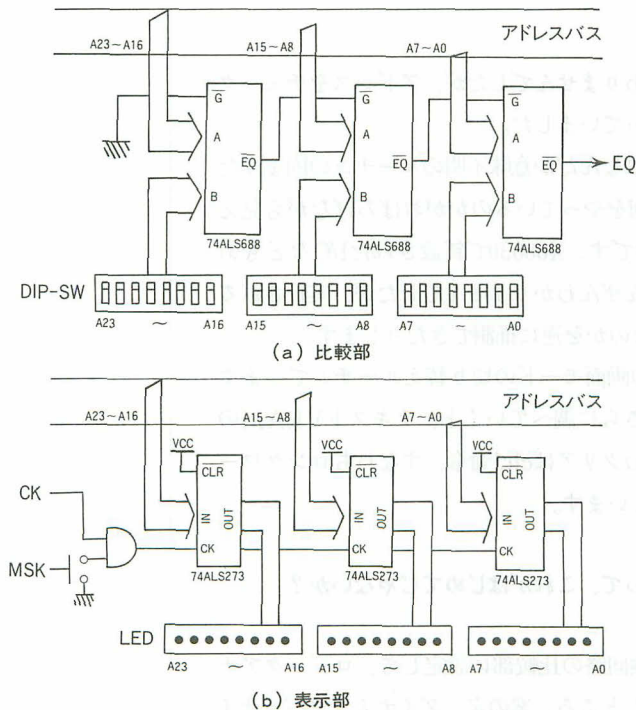



図3.2　アドレス線監視のための試験回路

ROMデバッガについてはマニュアル類にも書かれていないので、あまり知られていませんが、X68030のROMに内蔵されているデバッガプログラム*1があるのです。RS-232Cにつないだ端末から操作するようになっており、X68のキーボードや画面が使えないようなシステム自体のデバッグに威力を発揮します。これは、X68000のROMにも存在していました。機能的にはXCについてくるデバッガDB.Xとほぼ同じレベルですが、なんといってもHuman68kが立ち上がらない状態でも使えるため、「68020 on X68000」を作ったときは、このROMデバッガが重宝しました。

＊1
メモリスイッチの隠しオプションを指定すると、起動時に組み込まれ、エラーが発生したときやNMIボタンを押したときにROMデバッガに制御が移ります。

このため、今回もROMデバッガを使用する設定にしていたのですが、どうもX68030に搭載されている68030対応のROMデバッガは68040では使えないようです。

さっそく68030モードに戻し、switch.xを使ってROMデバッガをオフにするようにメモリスイッチの設定を変更*2し、68040モードにしてリセットをかけます。

＊2
switch db＝offで設定できます。

これで動くか？

やっぱり動かないことに変わりはありませんでしたが、アドレスをチェックすると、前よりも少し進むようになっていました。

そうこうやっているうちに、最初はなんだか意味不明のルーチンの固まりだったROMのプログラムでしたが、何をやっているのかがおぼろげながら見えてきました。こうなればしめたものです。X68030で新設されたI/Oなどもあって、最初はどんな働きのものかぜんぜんわかりませんでしたが、実行される処理内容からどのような目的のI/Oなのかを逆に推測できたりします。

そして、今度は、どうやらIOCSの画面モードの切り替えルーチンでつまずいていることがわかってきました。さらに調べていくと、テキストVRAMのクリアで失敗しているようです。このクリアはclr.l命令、すなわちロングワードでテキストVRAMをアクセスしています。

げ、ダイナミックバスサイジングって、これがはじめてじゃないか？

テキストVRAMのアドレスを試験回路の比較部に設定して、ロジックアナライザでアクセスシーケンスを調べたところ、案の定、ダイナミックバスサイジングの変換回路にミスを発見しました。

今度こそどうだ！

祈る思いで電源をオンすると、X68000時代から使っているディスプレイテレビ「CZ-600D」が高解像度モードに切り替わるときのリレーの「カチッ」という音とともに、画面には懐かしいIPL画面らしきものが表示されました。

実際には、文字は欠けてるわ、ゴミは出てるわで、まだまだ正常にはテキストVRAMアクセスができていない様子でしたが、やっと、目に見える形の変化が現れたわけです。

さらに、その後もいろいろと手を加えていくと、IPL画面の表示がだんだんまともになっていき、たまにフロッピーをガガッとアクセスする音がするようになりました。もっとも、その後、「エラーが発生しました。リセットしてください」というメッセージが表示され、そのうち黙り込んでしまうことに変わりはありません。

しかし、目に見える変化が出たことで、おおいに勇気づけられました。

Voice of Users

68040の鼓動

040turboを手にして最初に思ったのは、
「68040はやっぱり速かった！」
これです。

そのときは040SYSpatch.sysが68040の性能をまだ十分に引き出していなかったのですが、それでもやっぱり68030と比べると桁違いの速さでした。一昔前のワークステーションに積まれていたMPUを搭載したマシンを自分ひとりで占有できる幸せ（笑）をかみしめたりもしました。

040turboは、はじめのうちはまったく安定した動作をしてくれませんでしたが、今はゲームのとき以外は030モードにすることもないほど、040モードで安定して動いています。

実は、今でもまだ040turbo取り付け時のまま、X68030の側面カバーが開けっ放しなんです（笑）。ファンの大きな音を聞いて「040が動いてるんだなぁ」と実感しています。(^^;

（文●桃太郎　MAX-BBS MAX0896）

デバッグの日々　その3

それから、また、壁に突き当たってしまいました。

IPLの文字が欠けているのですから、VRAMアクセスにちょくちょく失敗していることは間違いありません。しかし、プログラムは結構まともに走っているようですから、ROMやメインメモリのDRAMのアクセスはうまくいっているようです。

X68000互換のI/O系の調子が悪いようです。しかし、具体的に何が悪いのかがわかりません。わりとまともに文字が表示されることもあれば、ボロボロのときもあります。とにかく試行錯誤でいろいろやってみるのですが、状況はあまり変わりません。

68040からX68000互換のI/O系をアクセスするのは無理なのか？

今度こそ、68040の単独動作をあきらめ、68030で運用して特定のアプリケーションだけ68040で走らせるという方法に逃げようかと考えました。プロセッサを自由に切り替えることができればなんとかなりそうです。しかし、いろいろ試してみましたが、どうもうまく切り替わってくれません。それに、68040自体の動作が不安定でした。万策尽きて、もう諦めの境地に達した頃、ほとんどヤケでマザーボードに手をつけることにしました。

実は、この段階では68040のPCLK（Processor Clock）信号に供給すべき50MHzを、25MHzのクロックから図3.3.aのような簡単な回路で作り出していました。

（a）回路イメージ　　（b）タイミング

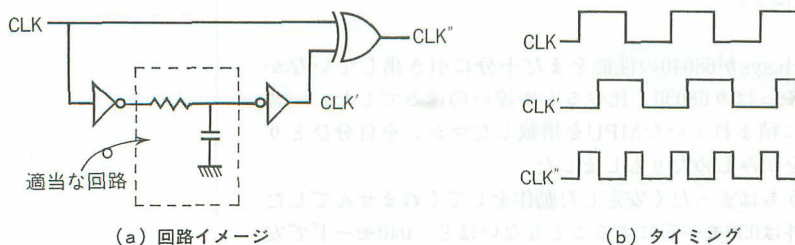



図3.3　簡単な倍クロック化回路

ゲートを何段も通して元の信号を遅らせ、図3.3bのようにちょうど10nsec遅れた信号を作り、元の信号のEOR（排他的論理和）をとれば、周期が20nsec、つまり50MHzの信号が取り出せるわけです。何段通すかはディップスイッチで設定するわけですが、ちょうど20nsec遅れるような、虫のいい波形は得られませんから、きれいな50MHzにはなりません。抵抗やコンデンサの値を調整して、試行錯誤で68040が動いてくれるようにしたわけです。

X68030のマザーボードを見ると50MHzのオシレータが載っていますが、68030は25MHzのクロックしか使いませんから、X68030の68030ソケットには、この50MHzを2分周した25MHzのクロックしか接続されていません。このため、50MHzを作り出す必要があったのです。

こんな面倒なことをして50MHzのクロックを作っている*1のも、X68030のマザーボードはきれいなままにしておき、68040の接続が失敗したら、いつでも元の状態に戻せるようにしておきたかったからです。

X68030のマザーボードに手をつけ、50MHzのクロックを取り出すことを決意しました。これをやっても動くかどうかはわかりませんが、もうほかにやれることを思いつきません。ここまできたら、もうヤケクソです。

＊1
モトローラではクロックを倍化する専用LSIを用意していますが、普通のパーツ屋では扱っていないようで、入手することができませんでした。

Voice of Users

私は人柱

ある日、040turboが届いた。さっそく取り付けてみた。おもむろに起動し、picファイルを表示させてみた。すると、こともあろうにグラフィックVRAMにゴミが出るではないか！

「DMAだけでなく、グラフィックVRAMまでもが根性なし*2なのか？」

何かの常駐物との相性が悪いのかとも思った。しかし、それを特定することはできなかった。

ある日、どうもゴミは実行アドレスに依存しているらしいということに気がついた。しかし、そのような報告をNIFTY-Serveにした人はいない。まさか68040自体のマスクの問題じゃぁ？　不安の日々*3。

ある日、BEEPs氏がゴミ問題対処の方法をアドバイスしてくれた。やってみた。グラフィックがモザイクになった。……

ある日、どうもこの現象はうちの機械だけでなく、多くの040turboユーザーのところでも発生するということが判明。BEEPs氏が040turbo側で対処方法を考えてくれることになった。至上の喜びであった。現在、私はその対処で問題ないか調べる人柱……。　（文●おゆ　NIFTY-Serve GBD02245）

＊2
うちの機械のDMAはクロック17.4MHzでさえコケる根性なし。

＊3
だって、68040って、高いんだもの。

オシレータの足は基板の裏側にしか見えていないので、どこか表側に出ていないかとオシレータ出力の配線をたどっていって、74F804というICの８番ピンにたどりつきました。これがソケットになっていてくれたら話は簡単なのですが、マザーボードに直付けする部品です。唯一の救いは、高密度のフラットパッケージが多用されているX68030のマザーボードのなかで、このICだけは普通のDIPタイプということです。このICの足に、写真3.2のように電線を直接半田付けして50MHzのクロックを取り出し、68040のPCLKにつなぎます。

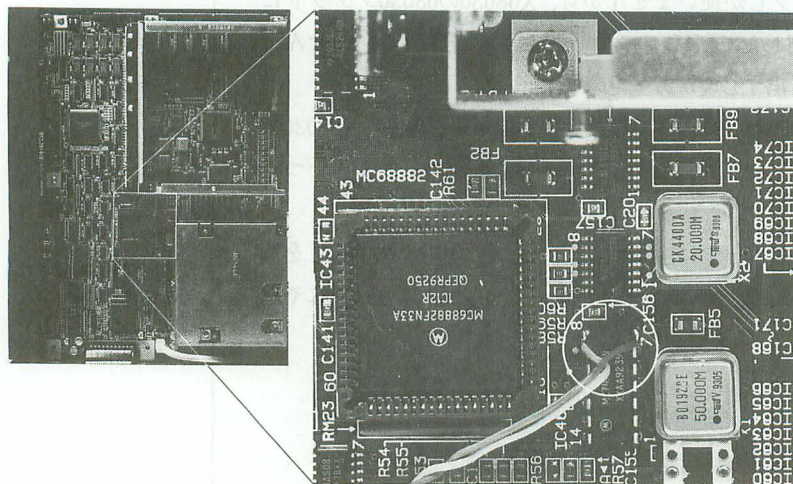



写真3.2　50MHzクロックの取り出し

これならどうだっ、とほとんど諦めの境地で、68040モードオン！

すると、どうでしょう。IPLの文字が欠けることなく表示されるようになりました。残念ながら、Human68kは起動しませんでしたが、エラーメッセージが表示されるまで結構長い時間、フロッピーをアクセスするようになりました。しかし、後、もう一息というところで、「エラーが発生しました。リセットして下さい」と表示されてしまいます。

その夜はもう遅かったので、これで終わりにしようと電源を切り、ロジックアナライザのホストであるPC-9801U２の電源も落としました。ここで、ふとHuman68kに処理が移ってコケているのかもしれないと思い立ち、Human68kの解析のため、逆アセンブルだけでもしておこうと、68030モードにしてX68030をパワーオンしました。

起動しない。

マザーボードに手をつけた直後のこと、ついに壊したかと、一瞬焦りました

が、これはロジックアナライザのプローブをつなぎっぱなしにしていたのが原因でした。実は、このロジックアナライザは、ホストであるPC-9801側の電源が入っていないと、つないだプローブがターゲットマシンの動作に影響を及ぼすのです。いつもはここでPC-9801U 2の電源を入れるのですが、Human68kの解析は結構時間がかかりそうなので、しばらくX68030ばかりを使うことになるだろうと思い、プローブを全部外しました*1。

無事、68030モードで起動し、逆アセンブルをすませた後、最後にもう一度、68040の状況を見ておこうと、68040モードにスイッチを切り替えてリセットしました。すると、フロッピーがガガッと音を立て、いったん止まってしばらくたった後、またフロッピーが回り出し、ズッズッズッズッと小気味よいアクセス音がした後、待望のメッセージ

```
Human68k ver3.01
```

Human68kの起動メッセージが表示されたのです。

＊1
プローブをつなぐと負荷も増えるし、ノイズにやられやすくなります。

Voice of Users

68040とED.R

68040という石はおろか、68030ですら私にははじめてだったので最初のうちは苦労しました。

実をいうと、040turboを注文した時点では私はPROユーザーで、X68030を買うお金すらなかったぐらいですから。どっちかというと040turboは衝動買いに近いものでした（笑）。後でゲーム目的でX68030を買った友人から、X68030を格安で譲ってもらいました（おかげで、夏休みのバイト代がすべて吹っ飛んだという話もあるが）。

さて、私はED.RというED.Xに似たエディタを制作しているのですが、そのプログラムは悪い見本といいますか、とても68040では（68030でも）動くものではなく、自己書き換えなど当たり前というものでした。

それからいろいろ勉強しまして、自己書き換えの後はキャッシュをクリアするとか、たとえキャッシュオンの状態でも暴走だけは避けられるように自己書き換えの順番を工夫するなどの変更をしてなんとか動くようにはなったようです（自己書き換えを止めろというのがもっともな意見かも……）。

（文●上田　智　NIFTY-Serve GBD03624）

Human68k

待望のメッセージは表示されましたが、途中でコケて〝#〟のプロンプト状態に落ちました。普通はエラーになると「リセットして下さい」といってくるので、このプロンプトを目にすることはないのですが、何か変なことをしてcommand.xがコケたり、command.xを削除して起動できないようにしておくと、この状態になります。

Human68k自体のコマンド入力状態のようで、command.xの内部コマンドであるdirやtypeなどは実行できませんが、外部のプログラム類をフルパスで指定してやれば実行することができます。

ここらへんは、ROM解析やらハード工作やら、さんざん普通じゃないことをやってきたので、だいたいは経験済みです。

ためしに「command.x」とタイプしてみましたが、やはりバスエラーで〝#〟プロンプトに落ちてしまいました。しかし、まだ暴走はしていません。いちおう、カーソルは点滅し、キー入力も受け付けてくれます。

COLUMN

X68000のROM-OS

これは、X68000を使いはじめの頃、ROMの解析をしていて偶然発見したものです。

マニュアルには一言も書いてありませんが、初代のX68000にはROMディスクというものが存在し、ここにHuman68kとTERM.Xが入っていました。そして、メモリスイッチの起動をROM 1 *1にすると、ROMのHuman68kが起動するのです。SRAMをSRAMDISKに設定しておくと、そのなかのcommand.xを実行しようとし、見つけられないと〝#〟プロンプトに落ちます。

自作のプログラムgame.x *2をcommand.xにリネームして入れておき、ハードディスクもフロッピーもアクセスせずに、いきなりgame.xが起動する様を見て、game専用マシンだと喜んでいたものです。

ちなみに、SRAMをNo_useにしておけば、ROMディスクのなかのTERM.Xが起動しターミナル専用マシンに早変わりというわけで、「隠し」にしておくにはもったいない機能でした。

＊1
今のswitch.xでは設定できません。

＊2
知る人ぞ知る、マイコン黄金時代にはやったgame言語のインタプリタのX68版です。

しかし、Human68kが起動しているのを見ても、68040で動いているという自信がありません。切り替えスイッチは、確かに68040モードですが、

スイッチが壊れたんじゃないか？
変換回路のモード切り替えロジックをミスっただけじゃないか？

プローブを外してしまったので、ハード的に68040の動作を確認することができません。一度電源を切って、ロジックアナライザのプローブをつなぎたい衝動にかられましたが、止めると二度と動かないかもしれません。

デバッガならどうだ？

っと、さっそく「¥bin¥db.x」[*1]とタイプしてみましたが、こちらもバスエラーです。

＊1
〝#〞プロンプトでは環境変数PATHなんてないので、フルパスで入力します。

なんとか簡単に確認できないか？

思い立ったのが、cache.xを利用することでした。

```
#¥bin¥cache.x
命令キャッシュ...........OFF
データキャッシュ.........OFF
```

なんとか、cache.xはエラーにならずに動くようです。それでは、とキャッシュをオンにしてみます。

```
#¥bin¥cache.x on
命令キャッシュ...........OFF
データキャッシュ.........OFF
```

おっ、キャッシュオンにならない。

これは、期待大です。というのも、68040と68030はキャッシュ制御レジスタのビット位置が変わっており、68030の制御ビットに当たるところは68040ではつねに〝0〞になっています。cache.xは、68030のつもりでキャッシュをオンにしようと制御ビットに〝1〞をセットしますが、〝1〞が書き込まれ

ないというわけです。このため、そこが〝0〟になっているので、68030のキャッシュ制御レジスタのつもりで見ると、つねにオフになって見えてしまい、cache.xでオンにできません。これが68040の証というわけです。

うれしくなって何度も実行しているうちに、とうとう暴走してキー入力を受け付けなくなってしまいました。キーボードリセットもききません。

ロジックアナライザをつないでないので無限ループに陥っているのか、ダンマリになっているのかさえもわかりません。

幻じゃないよな。

祈る思いで、もう一度リセットボタンを押してみます。

再びHuman68kのメッセージが表示されましたが、今度はデバイスドライバの登録のメッセージのあたりで、

バスエラーが発生しました。リセットしてください

となってしまいました。

しかし、とにかく、Human68kのブートには成功しているようです。

デバイスドライバを外して、原因を突き止めたい気持ちでしたが、すでに夜中の3時を回っており、明日も仕事があるので、パワースイッチをオフにします。リセットボタンを押すと、電源が落ちてくれました。

間違いない。68040でHuman68kが起動した。

とっくにカミさんは寝ていましたが、わざわざ起こして缶ビールの乾杯につきあわせます。ちょっと興奮ぎみにハードウェアの話をするのを見て、よくわからないながらもカミさんも手応えを感じてくれたようでした。ここ2ヵ月間というもの、家族を顧みないでやってきたことがやっと報われたわけです。もっとも、「次はソフトウェア、まだまだ先は長いぞ」というと、ガックリしていましたが。

まだまだ先は長いけど、
ひとまず、乾杯！
ビールがうまい！

歪んだ信号

結局、Human68kが起動するようになったのはクロックの安定化によるところが大きいのでしょう。ためしに取り出した50MHzのクロック線にプローブをつないだら*1起動しなくなってしまいました。それだけ、シビアというわけです。

ちなみに、040turboの配布基板でも、50MHzのクロックはX68030のマザーボードから取り出しています。ただ、ICの足へ半田付けするのはさすがに忍びないのでICクリップをかませる*2ようにしました。不安定になるかと心配したのですが、結構安定して動いてくれました。もっとも、クロックアップしているマシンでは、半田付けしたうえ、電線を極力短くしないと安定しないようです。

頭でデジタル回路を考えるときは、HighレベルとLowレベルだけの波形を思い浮かべますが、実際にはかなり歪んだ形になります。50MHzといったら相当に高い周波数ですし、今回の50MHzは、バッファも通っていない、オシレータの出力そのままです。

＊1
プローブ自体が負荷になるために、信号が鈍るのです。普通、こういうレベルで動作が変になってしまうのはマズイのですが、しかたがありません。

＊2
ICクリップでは接触不良を起こす可能性もあるので、いまいち心配だったのですが、25MHzで使う分にはおおむねいいようです。

Voice of Users

お気楽、極楽040turboライフのすすめ

私の040turboですが、購入以来、34.8MHzでしっかり稼働しています。

でも、最初はやはり不安定で、最初に取り付けたときなどは画面表示が乱れに乱れて起動すらしませんでした。040turboが届く前から、私のX68030はクロックアップ改造済みだったので、それが原因で040turboもイカれてしまったのではないかと思っていました。

しかし、多くの040turboユーザーの方々から情報をいただき、50MHzのクロックを取り出す配線を極力短くし、ICクリップもやめて、直接マザーボードに半田付けをしたところ無事起動!!　今では快適な68ライフを送らせていただいております。

今では、vdtファイルをサクサク特殊再生させたり、SX-WINDOWで窓をいくつも開いてサクサク動作する様を見ていると、生きててよかったと感じずにはいられません。SX-WINDOWにEGWordが乗ろうとも、きっと040turboならびくともしないでしょう。日頃X68030のスピードで悩み多きユーザーの方々が040turboを購入されて、お気楽、極楽な？68ライフを満喫してくださることを願ってやみません。

（文●よっち☆　NIFTY-Serve JBG03507）

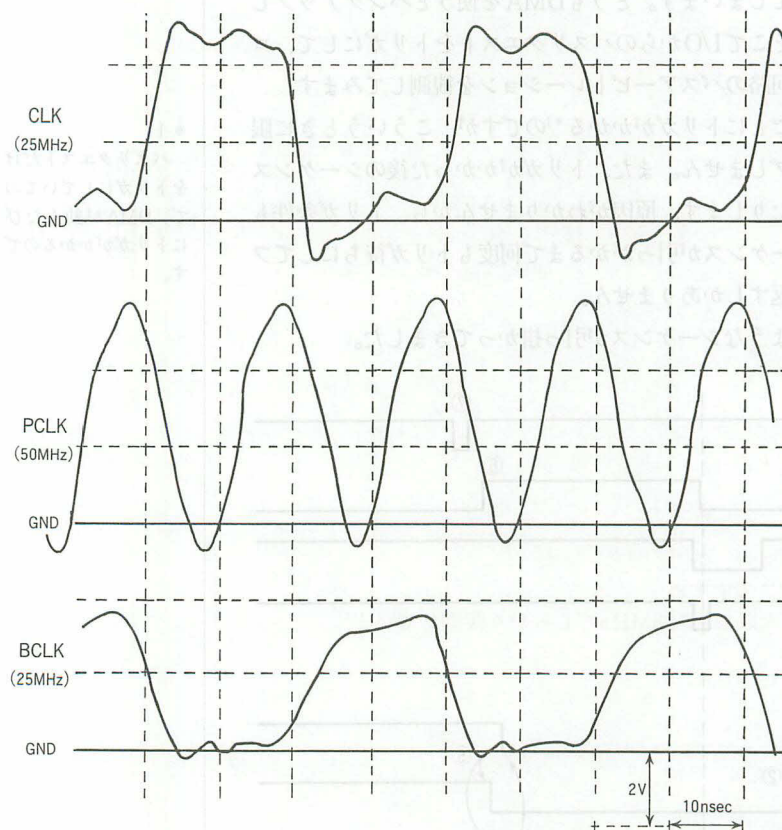
私はオシロスコープを持っていないため、実際の波形がどうなっているのか、このときはまだ見ていませんでしたが、ボロボロに歪んでいるだろうということは予測していました。後になって、040turbo配布の参加者でオシロスコープを持っている人がいたので、評価基板を送って波形観測をしてもらいました。図3.4が波形のスケッチです。バラック基板が動いていないときにこの波形を見ていたら、「こんなに歪んでいるんじゃあ、動かすのは無理*1だ」と、さっさとあきらめていたでしょう。

＊1
今でも、よく動いているなあと感心してしまいます。68040は、内部で波形を整形して使っているようなので、かろうじて動いているのでしょう。

図3.4　クロック波形

68040のバグ？

さて、クロックの安定化でHuman68kは起動するようになり、何回かに1回はcommand.xまで行くようになりました。しかし、デバイスドライバの登録でバスエラーになったり、うまく起動してもフロッピーのアクセス中にちょくちょくハングアップしてしまいます。どうもDMAを使うとハングアップしやすいような感じです。そこでI/Oからのバスリクエストをトリガにして、ロジックアナライザで変換回路のバスアービトレーションを観測してみます。

フロッピーのアクセスごとにトリガがかかる*1のですが、こういうときに限ってなかなかハングアップしません。また、トリガがかかった後のシーケンスでハングアップが起こったりします。原因がわかりませんから、トリガ条件も絞れず、ただただ異常シーケンスが引っかかるまで何度もトリガ待ちにしてフロッピーアクセスを繰り返すしかありません。

そして、ついに図3.5のようなシーケンスが引っ掛かってきました。

*1 バスリクエストだけをトリガにしているので、DMAが動くたびにトリガがかかるのです。

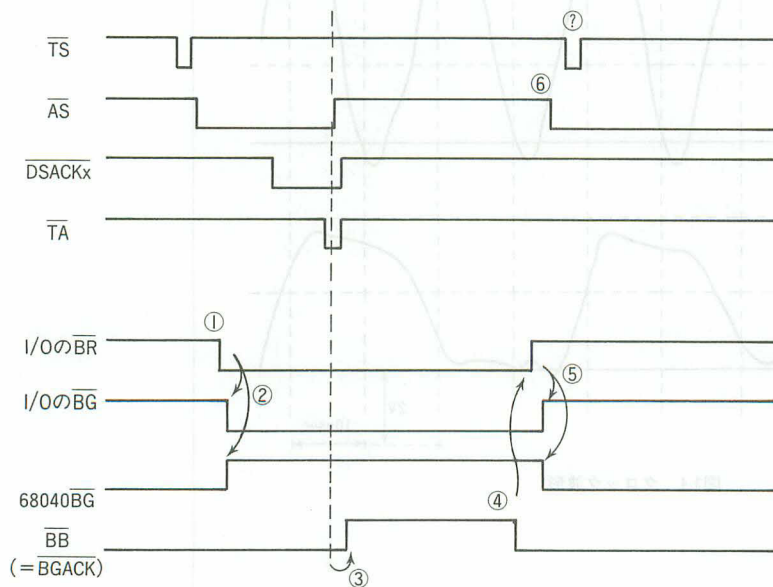



図3.5　フロッピーのアクセスエラーになるシーケンス

本来の68040のバス制御は、次のようになるはずです。

1. バスを使いたいI/OがBR信号をアサートする。
2. これを受けて変換回路は68040へのBG信号をネゲートする。
   同時に、I/OへのBG信号もアサートする。
3. 68040は現在のアクセスを終了すると、バスを開放し、バスを使っていることを示すBB信号（=BGACK信号）*1をネゲートする。
4. I/OはBGACK信号がネゲートされたのを確認すると、今度は自分がバスを使うことを伝えるためにBGACK信号をアサートする。
   同時にI/OはBR信号をネゲートする。
5. これを受けて、変換回路はI/OへのBG信号をネゲートする。
   同時に、68040へのBG信号もアサートする。
6. 68040はBB信号がネゲートされるのを待つ。
7. I/Oはバスの使用を終えると、BGACK信号をネゲートする。
8. 68040がバスを獲得し、BB信号をアサートする。

こうなるはずなのですが、異常シーケンスを見てみると、図の③の部分で、68040がBB信号をネゲートしたので、I/Oが「よしバスを使うぞっ」とBB信号をアサートしたのにもかかわらず、図の⑦の部分で68040からTS信号が出ています。

68040はバスの使用権を手放した*2にもかかわらず、平然と割り込んできて、I/O、この場合は、DMAの邪魔をしていたのです。

これは、どう見ても68040の動作がおかしいとしか思えません。使っているのがジャンク品の68040だけに、チップ自体にバグがあるのかもしれません。しかし、これがどうにかならないとフロッピーを含め、DMAが使えないことになります。試行錯誤でいろいろやってみると、68040へのBG信号をネゲートして実際に68040がアクセスをやめるのを待ち、さらに1クロックおいてからI/OへBG信号をアサートするようにしたら、なんとか使えるようになりました。

しかし、これまた運よく対処できたからよかったようなものの、できなかったらDMAをあきらめなければならなかったかもしれません。

そう思うと、バラック基板が動いたのはラッキーというより奇跡に近いと思えてきます。

*1
68040のBB信号は、X68030側のBGACK信号と直結しています。

*2
バスを使い終わったので、BB信号をネゲートしたはずなのです。

ベンチマーク

68040でHuman68kが動き出すようになると、試してみたくなるのがベンチマークテストです。もともと68040を動かすこと自体が目的のようなもので、性能は二の次と思ってきましたが、動くとなると気になってきます。

果たして、68040はどれくらいのスピードなのか？

ベンチマークテストといってもいろいろありますが、とりあえず、手元にあったpower.x[*1]というフリーソフトを試してみました。これは、X68000用の性能測定プログラムとして有名なものの1つです。10MHzのX68000のスピードを基準[*2]に、相対性能がパーセント表示されるものです。

さて、これで測定してみた結果が表3.1です。68040の性能はわずか113.98%、キャッシュをオンにできていないということもあるでしょうが、68030のキャッシュオフの性能と比べても芳しくありません。しかし、68030のキャッシュオン時の性能も、173.75%とこれまた低すぎます。

後でわかったことですが、power.xは単位時間に何回ループを回ることができるかで性能を計測しているのですが、この単位時間のチェックにRTC(Real Time Clock)をアクセスして秒のデータが変化するまで、つまり1秒経過するのを待っているのです。X68000XVI以降のマシンでは高速化に対応してI/Oアクセスにウェイトが入るようになっているため、相対的にループできる回数が減って値が低めに出るというわけでした。

しかし、このときはこのような詳しい事情を知りませんでした。とにかく、ちゃんとプロセッサのパワーを測定できるプログラムがほしかったので、友人

＊1
作者milk氏。

＊2
常駐ソフトウェアの負荷を見るために、実行したマシンの無負荷状態からの相対性能を表示するバージョンもあります。これだと、性能表示が100%を超えることはありません。

表3.1 power.xの結果

マシン	プロセッサ	クロック	キャッシュ	power値
X68000	68000	10MHz	なし	97.84%
68020 on X68000[*3]	68020	10MHz	オフ	78.13%
68020 on X68000[*3]	68020	10MHz	オン（命令）	172.71%
X68030	68030	25MHz	オフ	173.56%
X68030	68030	25MHz	オン（命令／データ）	173.75%
68040 on X68030	68040	25MHz	オフ	113.98%

注：X68030はスタティックカラムモードオフで使用。

＊3
前に作ったボードです。

のYakkun氏[*1]に頼ることにしました。彼のX68000XVIは24MHzにクロックアップされていますし、草の根ネットもこまめにチェックしているので、他の性能測定プログラムも持っていそうです。

時間は夜の11時を回っていましたが、かまわず電話をかけてしまいます。

私：クロックアップしたマシンで性能測定するベンチマークプログラム、何か持ってたら送ってよ。

これで、ピーンときた[*2]ようです。

彼：え、68040って、もうベンチマークやるレベルになったの？
私：まだ若干あやしいけど、Humanが動くようになったんだよ。
彼：おっほーッ、すごいじゃん。

結果が出たら知らせるから……、そんな約束をして電話を切りました。夜中の２時頃、まだ無理かなと思いながらもNIFTY-Serveにアクセスしてみると、彼からプログラムが入ったメールが届いていました。そのうちの１つがpv.xでした。

pv.x[*3]は、power.xの不具合であるI/Oアクセスのウェイトを避けるためにタイマ割り込みが使われており、ほぼプロセッサの速度比[*4]が反映されます。

もう寝るつもりでしたが、せっかくプログラムが届いたのです。試してみないと気がすみません。今度こそ、と勇んで実行してみましたが、やはり68040の性能は、表3.2のとおり芳ばしくないものでした。

今度は、X68030のキャッシュオン性能が初代X68000比5.48倍ですから、いい線です。プログラムの問題はないとなると、68040の1.98倍という数字を信じるしかありません。キャッシュオフとはいえ、これは、期待外れのものでした。

表3.2　pv.xの結果１

マシン	プロセッサ	クロック	キャッシュ	X68000比
X68000	68000	10MHz	なし	0.98倍
68020 on X68000	68020	10MHz	オフ	0.76倍
68020 on X68000	68020	10MHz	オン（命令）	1.73倍
X68030	68030	25MHz	オフ	2.50倍
X68030	68030	25MHz	オン（命令／データ）	5.48倍
68040 on X68030	68040	25MHz	オフ	1.98倍

注：X68030はスタティックカラムモードオフで使用。

＊1
ちなみに、彼は私と同期入社だったのが腐れ縁の始まりで、X68000の道に引っぱり込まれ、パソコン通信にもはまり、今度は040turboにも参加して（させられて？）います。

＊2
実は、この040turbo計画、うまくいかなかったら恥ずかしいので、人にはほとんどいっていなかったのですが、１回だけ、メモリへのアクセスに成功した頃に（我慢できなくなって）「こんなことをやってるんだよーん」と、MAX-BBSという草の根ネットでほのめかしたことがありました。

＊3
作者K.Nakayama氏。

＊4
といっても、単純な命令の繰り返し回数ですから、いつもこの数値のような性能が出るわけではありません。

かつて「68020 on X68000」を作ったときは、作る前からあまり性能が上がらないという噂を聞いていましたので、性能がふるわなくても*1そんなにショックを受けませんでした。

しかし、今回のX68030への68040の搭載はまだ誰もやったことがないはずです。68040の潜在的なパフォーマンスからすれば、変換のオーバーヘッドを考えても68030の２倍はいくと期待していたので、キャッシュオフ状態の比較とはいえ、68030より劣る数字が出たというのはショックでした。

＊１
power.xやpv.xの評価は単純ループなので７割増しですが、実際のアプリケーションでは２割アップがいいところ。これではX68000 XVIにもかないません。

68040が、どうして、こんな性能しか出せないの？

こんな結果では恥ずかしくて表に出せない。また、お蔵入りか？

しかし、まだ望みがなくなったわけではありません。68040には68030の16倍の容量のキャッシュがあります。キャッシュオンにすれば、逆転できるかもしれません。なんとかキャッシュオンに挑戦してみます。

ああ、キャッシュオン

cache.xで68040キャッシュをオンにできないのは、キャッシュ制御レジスタのキャッシュイネーブルビットの位置が68030と68040では違っているためです。それならと、デバッガから直接キャッシュ制御レジスタを叩いて該当ビットをオンにしてやります。しかし、デバッガ自体がハングアップしてしまいました。あわてて68040のユーザーズマニュアルを調べると、68040はリセットしてもキャッシュ内容がクリアされないため、キャッシュをオンにする前はCINV（Cache INValid）命令を実行して、キャッシュをクリアしておかなければいけないと書いてありました。

ところが、68040のユーザーズマニュアルには、どこにもCINV命令のオペレーションコード、いわゆる16進数の機械語コードについての情報が記載されていないのです。68030のユーザーズマニュアルには各命令を詳細に説明しているページがあるのですが、このCINV命令は68040で新設された命令ですから載っていません。68040から新設された命令の詳細説明*1は「68000 PROGRAMMERS HANDBOOK」という別のマニュアルに分かれてしまっていたのでした。実際、68040のユーザーズマニュアルは68030のマニュアルの半分以下の薄さで、詳細説明は載っていないなとは思っていたのですが、命令の一覧表があったので、そこに機械語コードも書いてあると思い込んでいました。いずれはプログラマーズハンドブックのほうも入手するつもりでしたが、まさか、こんなに早く必要になるとは思っていませんでした。

キャッシュオフ時の不本意な数字のまま、Yakkun氏に言い訳まじりの報告メールを書いて、その日は4時頃、床につきました。

＊1
ちなみに、ハードウェアの詳細説明も「68000DESIGNERS HANDBOOK」という本に分かれています。

キャッシュオンのベンチマーク

とりあえずほしいのは、CINV命令の機械語コードです。翌日、知っていそうな知人に電子メールを出しまくりました。ほどなく2人からCINV命令のフォーマットについての返事が届きました。さらに、ついでにお願いしていたDHRYSTONEやWHETSTONE[*1]も入手することができました。

さっそく、デバッガでCINV命令の機械語コードと、キャッシュ制御レジスタを操作する機械語コードを書き込んで実行します。さすがに、今度はハングアップしません。それではと、デバッガを抜けてベンチマークプログラムを実行しようとすると、ここでバスエラーになってしまいました。

考えてみると、フロッピーのアクセスはDMAによって68040の知らないところでメモリ内容の書き換えをしていることになります。すると、プロセッサのキャッシュは古いデータになってしまいますから、DMA後にキャッシュをクリアしてやらなければなりません。IOCSやHuman68kのなかでは、このようなキャッシュクリアの処理もこまめに行われているのでしょうが、68040のキャッシュクリアの方法は68030のキャッシュクリアの方法とは違うので、キャッシュ内容がクリアされずにそのまま有効となり、キャッシュ上の変なプログラムやデータを実行しようとして暴走してしまうのでしょう。

最終的にはこれらの箇所をパッチしていくことになるわけですが、今はとにかく、ベンチマークプログラムをキャッシュオンで走らせるのが先決です。それならということで、デバッガで先にベンチマークプログラムを読み込んでおいてキャッシュをオンにします。そうしておいて、すでにメモリ上にローディングされているベンチマークプログラムのスタート番地から実行[*2]させてみたところ、めでたく実行させることができました。

結果は表3.3のとおりです。pv.xは、実に10MHzのX68000比で16倍、これには一瞬我が目を疑ってしまいました。X68030のキャッシュオンと比べても3倍です。

一気に逆転。そうよ、そうよ、こうでなくちゃ。

DHRYSTONEやWHETSTONEは、プログラムがすっぽりキャッシュに載ってしまうため、実際のアプリケーションの実行ではここまで顕著な効果はないかもしれませんが、それでもX68030の2倍から3倍の性能[*3]は叩き出

＊1 ベンチマークテストプログラムの定番です。

＊2 X68000上で68020のキャッシュをオンにしてテストしたときも、この技を使いました。

＊3 X68030の広告でも演算性能はDHRYSTONE比となっていますから、この比較は許されるでしょう。

してくれそうです。

まだ、キャッシュオンにするとファイルアクセスができなくなりますから、実用には程遠い段階ですが、頑張る価値は十分あります。それに、なんといっても胸を張って公開できる結果です。

表3.3　pv.xの結果2

マシン	プロセッサ	クロック	キャッシュ	X68000比
X68000	68000	10MHz	なし	0.98倍
X68020 on X68000	68020	10MHz	オフ	0.76倍
X68020 on X68000	68020	10MHz	オン	1.73倍
X68030	68030	25MHz	オフ	3.59倍
X68030	68030	25MHz	オン	5.50倍
X68040 on X68030	68040	25MHz	オフ	2.04倍
X68040 on X68030	68040	25MHz	ライトスルー	16.44倍
X68040 on X68030	68040	25MHz	コピーバック	16.37倍

Voice of Users

ベンチマークオタクの独り言

私は、コンピュータと名がつくものを見るとベンチマークを計りたくなる性分のようです。BEEPsさんが進められている040turbo計画への参加を表明したときに真っ先に聞いたのが、DHRYSTONE ver 2の入手方法でした。(^^;

てなわけで、040turboがきたときに計ったベンチマークの味は格別なものでした（なんだ、そりゃ）。結局、私は040turboの凄さに圧倒されて、使いこなしなんてことはもう全然別の次元の話で、いろんなソフトを走らせては「速い速い～」と、いまだに040turboの速さにニヤニヤしているような状態です。

040turboで私が苦労したことといえば、空冷ファンの取り付けに七転八倒したことぐらいです。今後参加される方は、040turboに取り付ける放熱ファンの高さに注意してください。下手なものを買うと、私みたいに結局2つ冷却ファンを買うことになります。

040turboの今後の課題があるとすれば、X68030本体の天板から出しているLEDと放熱ファンの電源コードをどうするかです。*1（かっちょわり～い）

（文●TeM NIFTY-Serve HGE02300）

＊1
放熱ファンが本体カバーに当たってカバーが閉まらなくなります。

040turbo第一報

パソコン通信で書き込んだ040turbo＊1の第一報は、次の文章でした。書き込み先はもちろん、NIFTY-Serveのシャープユーザーズフォーラム

＊1
このときは、まだ040 turboと呼んでいませんでした。

097/999　PEG00631　BEEPs　　68040 on X68030動きました
(14) 93/07/06 23：24　　コメント数：6

RTで報告していましたが、68040 on X68030　の実験が一応の成果をおさめましたので、報告します。

X68030のIPLは、68040には対応しておらず、68030という表示になり、クロックの表示も正しくないですが、FDからHuman68k Ver3.1は起動でき、主なコマンドは実行できるようになりました。ただ、SCSIからの起動は、エラーとなりできません（SCSIDRV組み込みで、アクセスはできます。）

性能は、以下の通りとなりました。

	68000	68030		68040		
		cache off/on		cache off/wt(＊a)/cb(＊b)		
pv.x(＊1)	-	3.59	5.50	2.04	16.44	16.37
dhrystone（＊2）	1529	4854	6578	5208	16666	25000
whetstone（＊3）	96.06	251.26	279.33	194.55	476.19	689.66

(＊a) ライトスルー・モード
(＊b) コピーバック・モード
(＊1) power view
(＊2) version2.1　500000回指定
(＊3) float 2 使用　浮動小数点演算をf命令で直接駆動すればもっと早いでしょう

大ざっぱに言って、キャシュオフの時は、68030よりも若干遅いですが、キャシユをオンにすると、68030の2倍以上の性能はでるようです。

なお、キャシュ制御が68030と68040とでは異なるため、キャシュオンのままHuman68kを使うとファイルアクセスなどで異常になる（キャシュクリアが失敗するのでしょう）ため、テストでは、db.xでターゲットプログラムをローディングした後、キャシュをオンで実行しています。
現在のハードは、バラックのドータボートをX68030のMPUのソケットを

介して接続し、オリジナルの68EC030および、68040と外付け回路を載せてます。もともと、ハード的に68040と68030は互換性が無いので、外付け回路で

1）外部アクセス信号およびタイミング変換
2）ダイナミックバスサイジングのサポート
3）バスアービトレーション制御
4）クロック関連

などを行ってます。回路規模は、GAL16V8×3個、GAL20V8×2個、74AS245×6個、74AS374×2個、74AS74×1個です。
なお、50MHzのクロックのみ、マザーボードの50MHzのクロックモジュールから直接取ってます（これはドータボード上で生成できるようにする予定）

さて、このような状況で、ハードは何とかメドがついてきました。そこで、プリント基板を起こそうと計画しています。

現状は、ソフト（主にHuman）に問題があるので、アプリケーションがバリバリに動くというわけにはいきませんので、どちらかというと、68040を使えるようにするための協力スタッフを募るという性格になります。

なお、プリント板だけの製造原価自体は、1～2万円程度ですが、設計費や版代等が結構かかる模様です。人数が見えてきたら、再度価格について報告します。ちなみに、68040は、現在、6万円以上します。
まあ、とにかく、興味のある方は、参加表明してください。

PEG00631　BEEPs（大塚）

FSHARP1「ハードウェアの部屋」とMAX-BBSのX68ボードです。また、他のネットにも転載されていきました。
次の日には、さっそくレスポンスがついていました。

098/999　JBB00531　ゆい　　　RE：68040 on X68030動きました
(14) 93/07/07 00：04　097へのコメント

おおぉぉ！私憧れの040が動いたのですか！私は、ずぶの初心者のため、何も協力出来ないとは思いますが
少しでも力になれることがありましたら（ないない（^^;）協力させて頂きたいと思います。（人柱．．いや、68柱（？）でも構いません。）

しかし、今月の「COMPUTER DESIGN誌」をみて、「(まともな性能を出す事の出来る) アクセラレータの制作は難しいだろうなぁ」と思っていたところドライで、25000とは．．．．

今後の展開が楽しみです。頑張ってください。

JBB00531：ゆい

099/999　GHH01360　伊寿墨眼仁奈　RE：68040 on X68030動きました
(14) 93/07/07 01：23　097へのコメント　コメント数：1

おお、040ですね。

Macだと、ほとんどのソフトがキャッシュオンで動くんですが。
(EXCEL2.2Jが動かなかった)

X68030持ってないんで訳にはたてませんが、記事を転載させて戴きますね (^^；)
レモン通信in広島
0823-20-0068です・。(宣伝宣伝 (^^;))

伊寿墨眼仁奈

103/999　JBG03507　大浦義宏　　RE.X68030で040を・・・
(14) 93/07/07 17：39　　コメント数：1

いゃー凄いですねぇー、誰かがやるとは思っていましたが、まさかこんなに早くできるとは・・・それにしてもキャッシュオン時の040パワーは炸裂してますねぇー。(凄いです) そうそう040が高速なのはいいのですが、周辺回路とのウエイトはいかほどのもんでしょうか？まぁ030と切り替えが可能という事で某所でやっている従来機種X68K用のアクセラレータ030の50MHzで68Kの2から3倍 (クロックMPUには50MHzで最高は80MHzで動いたそうです・・・すげぇー) との事ですから、その030の2倍といえばもう涙ものですね。あいにく僕はプログラムを組める才能もないし、ハードについても素人なおろか者なので、どうしょうもないですが、一応X030は買ったのでどうぞ人柱として扱き使ってやって下さい・・・(^^；) 今イメユニ2の為に貯金しているので資金はなんとかなりますので・・・。
それではこれからどうぞよろしくお願いいたします。

P.S.そういえば、家のX030は34.8MHzなんですが、IOデーター製のメモリボードで完動するのでしょうか？本体の方はジャンパー飛ばしてたけど。

大浦　義宏

104/99　9PAF03012　Arimac　RE：68040 on X68030動きました
(14) 93/07/07 22：23 097へのコメント

いや～、久々に血沸き肉踊る話題ですねー。(^^)(^^)(^^)
４倍速程度では面白くないと、焼き鳥には手を出していない私ですが、
流石に16倍速となると私財を投げ売ってでも手に入れたい。
(実数演算では焼き鳥でもGCC30で80倍速でたが…)

それと、メモリーが12MB以上増設出来れば何も言うことはないですね。
話がもう少し具体的になってきたら参加したいと思います。

Arimac

反応は上々です。それに、基板配布についても、値段も何もまったくわからないにもかかわらず、すぐに10人ほどが参加表明してくれました。

私と同じように68040に熱い思いを持っている人はまだまだいるようです。

プリント基板

とりあえずバラック基板でHuman68kが動くようにはなりましたが、キャッシュオンにするとファイルアクセスができなくなるなど、まだまだ解決しなければならない問題は山積みです。１人でやっていても限界があります。

ソフトウェアなら他の人に試してもらうのは簡単です。現にX68000の快適な環境を支えている多くのフリーソフトウェアがそうやって育ってきました。しかし、ハードウェアはそうはいきません。クロックアップのような簡単な回路ならまだしも、68040のバラック基板を作れる人はそうはいません。やはり、ちゃんとしたプリント基板を起こす必要があります。10人程度では基板の設計費のほうが高くつきそうですが、どの程度の値段になるか見通しがつけば、もっと多くの人の参加が期待できるかもしれません。

まだ完全とはいえないし、値段もわからないというのに、040turbo計画に興味を持って協力してもいいといってくれている人がいたのに勇気づけられ、ほとんど68040と心中覚悟でプリント基板の製作を決心しました。

基板の種類

さて、プリント基板ですが、バラック基板を作るように電線で1本ずつ配線するのであれば必要な端子を自由に結ぶことができますが、プリント基板は銅箔のパターンで回路が作られますから、同じ面で配線が交差することは許されません。このため、基板の裏と表といったように複数の面を使って、交差しないようにしてパターンを引き回さなければなりません。

４層基板というのは、図3.6.aのように、表と裏の２面に加え、基板の中にも２つの層を持つ基板です。合計４面のパターンがありますから、自由度はさらに高くなります。といっても、基板の中の２層は電源とグランドのそれぞれの専用パターンにするのが普通です。

このほかに図3.6.bのようなパターンの密度も基板の設計に影響します。普通のICのピンは0.1インチ間隔で並んでいますが、このピンとピンとの間に何本のパターンを通すかという意味で、ピン間２本とかピン間３本[*1]といってパターンの密度を表します。層数、ピン間の本数が多いほど、パターンの自由度は増え、より複雑な回路を実現することができるようになりますが、当然値段も高くなります。仕事でプリント基板を発注したことは何回かあるので、今回の基板は、だいたい４層基板、ピン間２本か３本でできるだろうと予想しました。

＊1
MZ-80Kのマザーボードはピン間１本の２層基板だったと思います。これに対し、X68030のマザーボードはピン間３本の６層基板でできています。

アートワーク

回路図からパターンを起こす作業を、アートワーク*1といいますが、040turboのレベルの基板となると、とても素人の手作業で作れるものではありません。

こういう作業のために、CADが使われます。また、アートワークにかぎっていえば、コンピュータに入力された回路図をもとにパターンを自動生成する「Auto Router」と呼ばれるプログラムなどもあります。しかし、個人で利用できそうな値段の、プリント基板設計用のCADはPC-9801やIBM PC/ATの世界を探しても見当たりません。あるのは２層のみピン間１本といった、おもちゃのようなものばかりです。

いろいろ聞いてみたところ、業者に基板の製造だけ頼むなら十数万円でできそうですが、アートワークも含めて依頼すると、50～60万円はかかるだろうとの話でした。よっぽど、自分でなんとかアートワークをやれないかと考えたのですが、サンハヤトのキットとは違います。極力シンプルな回路になるよう心掛けたつもりですが、68030と68040のピン数だけでべらぼうな数になります。紙と鉛筆で書けるようなものではありませんし、ミスったら元も子もありません。ここは、アートワークも含めて専門の業者に依頼するほうが賢明のようです。その分を基板の配布価格に上乗せしなければならないので、かなり割高になりますが、これはしかたありません。

いろいろとつてを当たって行って、ある業者に頼むことになりました。とりあえず使用部品のリストを渡して、おおまかな見積もりをとってみます。案の定、アートワークの設計費が半分以上を占めていました。

なお、最終的には見積もり時の数字より、だいぶ上がってしまいました。途中、部品の変更やら回路の修正やらといろいろあったのも響いていますが、この業者も最初は、個人から持ち込まれた回路がこんなに複雑だとは思わなかっ

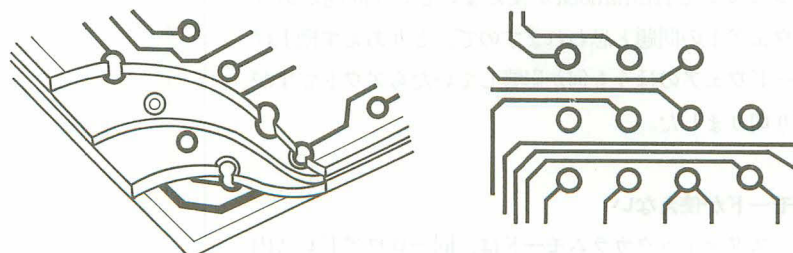
*1
パターン自体をアートワークと呼ぶ場合もあります。

（a）４層基板（概念図）　（b）ピン間３本のパターン

図3.6　プリント基板

たようです。

恥ずかしいことですが、変更につぐ変更のために、一番最初に書いた回路図のメモはグチャグチャで、見積もり時にはちゃんとした回路図が用意できなかったのでした。急いで清書して*1なんとか見せられるようになるまでに1週間以上を要しました。

＊1
この回路図は最終的にPostScriptファイルにしてNIFTY-Serve上にアップロードしました。

悩みの種

バラック基板でとりあえずHuman68kが動作することを確認したので、プリント基板を作ることを決意したのですが、まだいくつかハードウェア上の問題点がありました。一発で完全なものはできないと思っていますので、後から改造を加えなければならないのは覚悟の上です。しかし、アートワークに1ヵ月近くかかるということだったので、その間にできるかぎり解決して回路変更にまにあう分はアートワークに反映させたいところです。

その時点までに見つかっていた問題点としては、次のようなものがありました。

1）スタティックカラムモードが使えない
2）68882のアクセスが失敗する
3）ハードディスクから起動できない
4）グラフィックVRAMのアクセスが失敗する

このほかにも、キャッシュオンでHuman68kが使えないという問題がありますが、こちらはソフトウェア上の問題と思われますので、とりあえず棚上げしておきました。もしハードウェアのほうも何か影響していたらアウトですが、そのときはそのときと割り切りました。

1）スタティックカラムモードが使えない

前にも説明しましたが、スタティックカラムモードは、同一ロウアドレス内のメモリアクセスに関して最初の1回だけロウアドレスを与えて、以後はカラムアドレスを与えるだけで高速にアクセスができる方式です。スタティックカラムモードにするとアクセスが失敗するのは、68030と68040のアクセスのタイミングが微妙に違っているからでしょう。このまま「スタティックカラムモ

ードは使えません」という仕様にしてしまってもいいのですが、X68030のデフォルト動作はスタティックカラムモードです。マザーボードのジャンパースイッチSW1をショートしないと040turboが使えないというのは、問題を残したままという点でスッキリしません。

それにもう１点、なんとかスタティックカラムモードを使えるようにしたい理由があります。

68040はキャッシュをオンにすると、キャッシュの１ラインに相当する４ロングワード単位でメモリ読み込み*1をするようになります。68040は、この４ロングワードを高速にアクセスするために、バースト転送という特殊なアクセス方法を使おうとしますが、これは、X68030のメモリがサポートしていません*2ので、対応できません。

＊1
コピーバックモードにすると、書き込みに関してもライン単位になります。

＊2
変換回路で細工すれば対応できないことはありませんが、シンプルに作るという方針なのでやめました。

C O L U M N

バースト転送

バースト転送は、連続した４ロングワードのアクセスを前提として、特に高速にメモリをアクセスできるようにした特殊な転送方式です。68040は、最初のロングワードアクセスのときだけアドレスを出力しますが、残りに関してはいちいちアドレスを出力しませんし、TS信号も出しません。メモリからTA信号が４回返るのを待つだけです。

普通のロングワードアクセスが最少サイクルの２クロックですから、４ロングワードでは２×４の８クロックかかるのに対し、バースト転送では最初のアクセスの後は１クロックになって、２＋１×３の５クロックですみます。

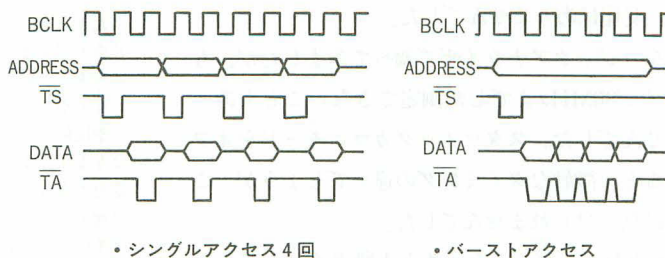



図　ノーマルの４回アクセスとバーストアクセスのタイミング

このため、変換回路で68040に対してTBI（Transfer Burst Inhibit）信号を返して、わざわざバースト転送を禁止しているのです。TBI信号を受けると、68040は１回ごとにアドレスを出力する通常アクセスになりますが、キャッシュの１ライン分を充塡するために４回のアクセス*1を連続して実行することに変わりはありません。このため、バーストモードほどの効率は期待できないにしても、連続アドレスのアクセスなので、スタティックカラムモードの高速アクセスは生きてきます。

また、Human68k側の問題でキャッシュをオンにできませんが、オンにできるようになればスタティックカラムモードが有利になります。Human68k側の対処はソフトウェアの問題ですから後からでもなんとかなるでしょうが、ハードウェアに問題を残してスタティックカラムモードが使えないままにしておくわけにはいきません。なんとかここで対処しておきたいところです。

さて、このスタティックカラムモードでアクセスが失敗するという問題ですが、最初はまったくRAMアクセスができなかったので禁止にしていましたが、Human68kも動くようになってきた時点でためしにスタティックカラムモードが使えるように戻してみました。すると、ほかの部分のハードウェアの動作が安定したということもあるのでしょうが、全然できないというわけではなく、デバイスドライバの登録あたりまでは進み、バスエラーで落ちました。相当に微妙なタイミングのようです。

そもそもX68030のメモリコントロールはカスタムLSI化されていて、どういうタイミングで動作しているのかがわからないため、外から見たアクセスタイミングから動作を推測するしかありません。68040側の信号の変換回路はこの推測をもとにしていますから、推測が間違っている可能性もあります。

*1
これを見てもわかるように、68040はバースト転送を前提としていて、バースト転送ができない場合というのは、あくまで例外的な位置付けなのです。

信号線が長いため、遅延が起こっているのか？

ためしに68040のアドレス線と制御線を、ドライブ能力の大きいラージバッファモード*2にしてみましたが、変化はありませんでした。

メモリアクセスのタイミングをロジックアナライザで調べてみましたが、もともと、このロジックアナライザが80MHzまでしか測定できないこともあって、微妙な部分がよくわかりませんでした。スタティックカラムモードをオフにすれば動くことも考えあわせると、微妙なタイミングの違いでしょうが、これが原因といえるような問題点は見つけられませんでした。

スピードが違うGALにかえてみたり、ゲートをわざと１段多く通してみたりと、いろいろやってみましたが、効果は上がりません。最終的に行き着いた

*2
大きな電流で信号をドライブしますが、信号パターンの設計をちゃんとやらないとノイズや反射を生みやすくなります。また、消費電力も大きくなるので、普通のスモールバッファモードで動けば、それにこしたことはありません。

のがウェイトの挿入でした。それも、アクセス開始前のウェイトという奇妙なものです。

普通、ウェイトというと、図3.7.aのように、データが返ってくるのに時間がかかり、プロセッサがデータを取り込むタイミングにまにあわないために、DSACKx信号を遅らせることをいうのですが、今回効果があったのは図3.7.bのように、変換回路で作り出すAS信号の出力を１クロック遅らせるというものでした。

この結果から逆に考えてみると、X68030のメモリコントローラはAS信号がアサートされる前からアドレスのデコードを開始してAS信号がアサートされたところでデコード済みになっていなければならないのに、68040からアドレスが伝わるタイミングが遅いのか、もしくは変換回路がAS信号を作るのが早すぎるために、デコードがまにあっていないのではないかと考えられます。スタティックカラムモードでは、ロウアドレスが同じかどうかでDRAMに対する制御のしかたをガラッと変えなければなりませんので、普通のDRAMアクセスよりタイミングが厳しくなっているのでしょう。

結局、68040からのアドレスの確定から、変換回路で作った68030もどきのAS信号のアサートの間に挿入した１クロックのおかげでタイミングに余裕をとることができたというわけです。DRAMアクセスにおいて１クロック余分に時間がかかるようになってしまいましたが、スタティックカラムモードが使えるので、平均時間としては早くなりました。

（a）普通のウェイト　　（b）前挿入のウェイト

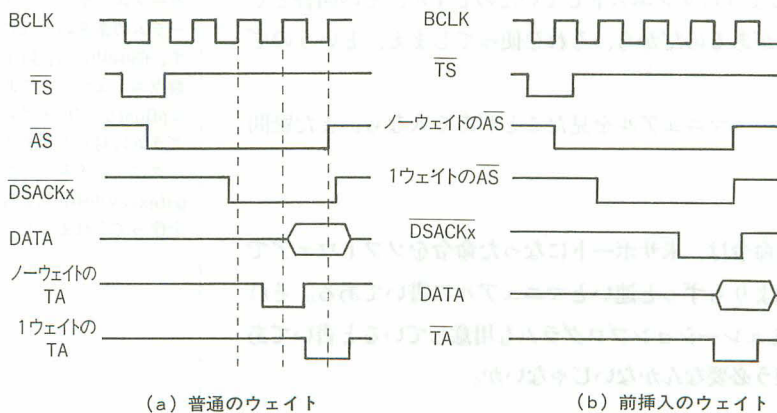



図3.7　040turboのウェイト１

2）68882のアクセスが失敗する

68882は浮動小数点演算プロセッサです。これを聞いて「アレッ」と思う人もいるでしょう。

68040は浮動小数点演算機能を内蔵しているので、浮動小数点演算プロセッサは不要。それどころかコプロセッサインタフェースをサポートしていないのだから、68882のアクセスはできないはずではないか。

確かにそうです。しかし、X68030自体は68030から使うための68882を搭載することができます。

コプロセッサインタフェースといってもコプロセッサとの間に専用の信号線があるわけではなく、68030からコプロセッサである68882の命令を使うと、68882を自動的にアクセスするシーケンスを実行するだけのことです。ただ、このアクセスはCPU空間という特殊なアドレス空間を使って行われているバスアクセスの１つ*1です。

このため、68040からでもmovesという命令を使ってCPU空間をアクセスし、68882へのコプロセッサインタフェースと同じシーケンスでアクセスしてやれば、68882を使うことは可能なはずです。

実は、この強引ともいえる方法は中村ちゃぷに氏*2の示唆によるものです。彼には、68040で未サポートになった浮動小数点演算命令のエミュレーションプログラムを作ってくれるようにリクエストしていたのですが、その回答として、どうせ68882そのものがあるのだから、それを使ってしまえ、というのでした。

ここで、68040のユーザーズマニュアルを見たことがある人なら、また疑問に思うでしょう。

68040の浮動小数点演算命令は、未サポートになった命令をソフトウェアでエミュレートしても68882よりもずっと速いとマニュアルに書いてある。それに、モトローラはそのエミュレーションプログラムも用意していると書いてあるぞ。わざわざ68882を使う必要なんかないじゃないか。

しかし、FPSPと呼ばれている、このソフトウェアパッケージ、どうも、結構なお値段のようで、今回のプリント基板の設計費どころの話じゃないという代物なんだそうです。このため、遅くてもいいから、とりあえず68882を使って動かせればいいじゃないという判断にもとづくものでした。そして、中村ち

＊1
ほかにCPU空間を使うアクセスとして割り込みアクノリッジサイクルがあります。

＊2
040turbo計画の参加者。NIFTY-Serveシャープユーザーズフォーラムの常連の１人です。68040用の浮動小数点エミュレータであるpfloatや、実行プログラムに対しダイナミックにパッチを当てるpatexecの68040対応版を作ってくれました。

ゃぷに氏はそのためのテストプログラムも送ってくれたのです。

さて、その結果ですが、68040ではそのテストプログラムは見事に失敗しました。

コプロセッサシーケンスを真似るのは無理なのかなあ。

ところが、このテストプログラムを68030で実行すると、ちゃんと期待したとおりに動きます。やはり、68040のアクセス自体に問題があるらしいのです。デバッガでトレースしてみると、最初の68882のステータスを読み出しているところですでにおかしくなっていることがわかりました。返ってくるデータが＄FFFFに化けていたため、ステータスのビジービットが立っていると誤認したのです。このため、68040はレディになるまでひたすらステータス読み出しを繰り返していたのです。

このときのアクセスタイミングをロジックアナライザで調べてみると、図3.8.aのようになっていました。案の定、68882からデータが返ってくるタイミングが若干遅いのです。とりあえずウェイトを挿入してみると、正常に読み出せるようになりました。今度は、アクセスの応答であるアクノリッジ信号を遅らせるタイプの通常のウェイトです。

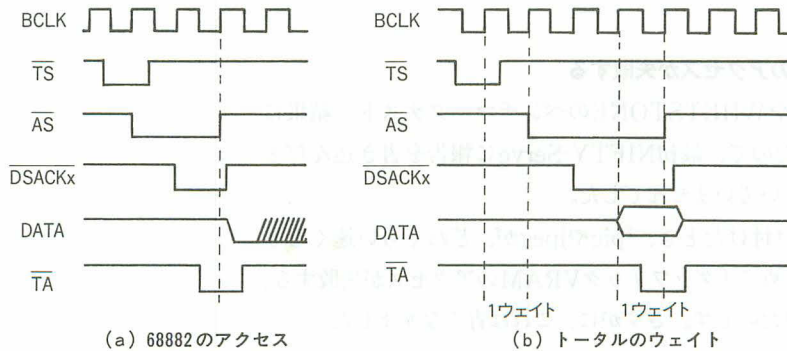
これで、68030からのアクセスよりもスタティックカラムモード実行のためのアクセス前のウェイトとあわせて図3.8.bのように合計２クロック[*1]も多くウェイトが入ることになってしまいました。

＊1
怪我の功名とでもいいましょうか、68040はウェイトが入って余裕ができた分、68030では通常動かないような高いクロックに改造しても、スタティックカラムモードで動かすことができたようです。

図3.8　040turboのウェイト２

3）ハードディスクから起動できない

この問題は、あまり深刻には考えてはいませんでした。まったくアクセスできなかったら困りものですが、フロッピーから起動させればちゃんとSCSIのハードディスクのアクセスもできるからです。起動時のSCSIデバイスの認識のところか、どこかほかのところで問題があるのでしょう。ソフトウェア的な問題のようにも思えましたし、私自身、SCSIインタフェースは苦手だったので、これまで後回しにしていたのです。

ブートROMの解析、ああ、SCSIも勉強しないといけないなあ。

そんなことを思いながら、メモリスイッチの起動デバイスをSCSI 0にして試したところ、あっさりSCSI 0のハードディスクから立ち上がってしまいました。

あれ、68040モードになってないのかな？

しかし、間違いなく68040が動作していました。結局、こいつはいつのまにか解決してしまいました。先のスタティックカラムモードと68882アクセスの対処で２ウェイト入るようになったのがよかったのでしょう。

ついでに、SCSI-IDの１番につないであるMOからの起動も試してみましたが、こちらもまったく問題ありません。何もしませんでしたが、この問題も片づきました。

4）グラフィックVRAMのアクセスが失敗する

実は、DHRYSTONEやWHETSTONEのベンチマークテストの結果に気をよくして浮かれていたので、最初NIFTY-Serveに報告を書き込んだときにはこの問題には気がついていませんでした。

基板の第一次配布を受け付けたとき、「picやjpegが、どれくらい速くなりますか」と聞かれて、はじめて「グラフィックVRAMのアクセスが失敗する」という重大な問題を見つけたのです。さすがに、これは青くなりました。

040turboでは、グラフィックが使えません。

いくらなんでも、これを仕様といって許してくれるX68000ユーザーはいないでしょう。あわててデバッグを行いました。

さて、実際の現象を調べてみると、picやjpegでグラフィックを表示しようとすると、いきなりバスエラーになってしまいます。最初は、これらのプログラムが高速化のために自己書き換えなどのプログラムテクニックを使っているのかなと思って調べていったのですが、なんとX-BASICからグラフィック画面を使おうとしてもバスエラーになるのです。

さらに調べてみると、グラフィック画面を表示しようとしただけでバスエラーになってしまうことがわかりました。command.xの内部コマンドであるSCREENコマンドを使って「SCREEN 1, 3, 1」としただけでも見事にバスエラーになります。これでは、互換性どころの話ではありません。

画面モードの変更でおかしくなっているようなので、CRTコントローラの設定のためのI/Oアクセスがまずいのかと考え、試験回路の比較部にCRTコントローラのアドレスを設定してロジックアナライザでアクセスシーケンスを追おうとしたのですが、引っかかってきません。しようがないので、DB.Xでpicファイル表示プログラムをステップ動作させて調べてみました。

デバッガによるステップ動作は、ハードウェアのデバッグにおいても威力を発揮します。現象が発生する直前まで動かしておいて、次に起こるというところでロジックアナライザを待機させることができるのです。ハードウェアオンリーの頃のデバッグと比べて、格段にデバッグ効率がよくなりました。

C　O　L　U　M　N

Human68k ver 3のDOSコールとDB.X

X68030になって困ったことは、DB.Xでデバッグしようとするとプログラムによっては特権命令違反になってデバッグできないものがあるということです。これは、Human68kのDOSコールで使っているFライン命令が、68030ではコプロセッサ命令となってしまい、ユーザーモードで実行するとFライン命令ではなく、特権命令違反になる番号ができてしまったからです。

Human68k ver 3では、特権命令違反のルーチンで、DOSコールだったら正規のルーチンを実行するようにしているので問題ないのですが、X68000のDB.Xは特権命令違反のベクタを書き換えて本当に特権命令違反が起こったものとして扱ってしまうのです。

しかし、68040ではコプロセッサをいっさいサポートしなくなったので、DOSコールにあたるFライン命令で特権命令違反になることはありません。このため、X68000時代のDB.Xがいちおう使えるのです。

ところで、68030でDOSコールがコプロセッサ命令と解釈されて問題を起こすために、問題となるDOSコールの番号自体を変更するという荒療治が行われましたが、68040の場合は、この問題は起こりません。この点からも、X68000の後継は68030でお茶を濁さず、68040にしたほうがよかったような気がします。

ただ、その分、デバッグに頭を使わなくなりました。デバッグ作業が難しいと、どういう現象で発生しているのか、あれこれ推理して、どうやったら調べられるか、一歩一歩詰めていくということをせざるを得ないので、結構頭を使います。

ところが、デバッグが簡単にできるようになると、あまり考えずに手当たり次第に実行してみることになります。しかし、的はずれなところを調べていると、いっこうに解決策にたどり着けません。

さらに運の悪いことに、DB.Xはバスエラーの発生した場所を正しく表示しない*1ということに気がつくのが遅れたせいで、関係がないところを調べては1人で混乱していました。

結局、この問題は、CRTコントローラを設定しているところではなく、グラフィックVRAMのアクセス自体に問題がありました。X68030は、X68000からの伝統で、リセットしてもグラフィックVRAMをクリアせず、グラフィック画面を表示しないようにするだけです。このため、画面モードを切り替えてグラフィック画面を表示するときになってはじめて、ゴミが表示されないようにVRAMをクリアするようになっているのです。040turboは、このVRAMアクセスでバスエラーを引き起こしていたのです。

ただ、この現象はかなり奇妙なものでした。図3.9のようにグラフィックVRAMのアクセスそれ自体は正常に終了するのですが、その直後の通常メモリのアクセスの最初でバスエラーが出てくるのです。もちろん、68030ではそんな現象は起こりません。これまた、メモリコントローラの内部やバスエラーを検出するタイミングについて詳細がわからないので、はっきりしたことはいえませんが、グラフィックVRAMアクセスのアドレスから通常のメモリのアクセスに切り替わるタイミングに何か問題があるのでしょう。アドレス信号線が変化するタイミングが早すぎてVRAMのバスエラー検出回路が誤動作しているのかもしれません。

現象がわかったところで対処方法を考えなければなりませんが、AS信号をアサートするタイミングやネゲートするタイミングを調整してみましたが、改善されません。

しようがないので、対処療法として、変換回路でAS信号をアサートした後、1クロックの間（図3.9の斜線部分）、バスエラーをマスクし、68040にエラーを返さないようにしました。

これもまた、一種のウェイト挿入なのですが、バスエラーがAS信号のアサートの直後に返ってくることは普通あり得ませんし、1クロック検出が遅れても実害はないでしょう。

＊1
これは、68000と68040ではエラーが起こったときにスタックに格納される情報の順序が違っているためです。そんなことは知らないDB.Xは、68000のつもりでエラー箇所を表示しようとしますので、68040が本来エラーを起こした場所とは違った場所を表示してしまうのです。

これで、グラフィック画面をアクセスできるようになり、はじめて68040によるpicファイルの表示を見ることができました。

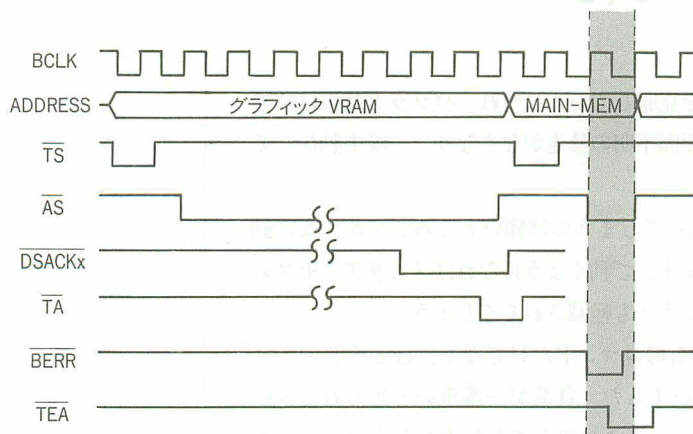



図3.9　グラフィックVRAMアクセスとバスエラー

Voice of Users

040turboはCGA野郎の必需品

040turboの御利益について、私の体験をお話ししましょう。

私の手もとには、MoonLigtR.CGAなる、４Mバイト強のCGAビジョン用のアニメーションファイルがあります。内容についてはご想像にお任せしますが（笑）、このデータを再生してみます。MPU68030では、こういう大きなデータをハードディスクから読み込みながら再生すると、ときどきディスクアクセスやデータ展開のためか、数秒間画像が停止してしまい、「ああっ、美奈子ちゃんのポーズが飛ばされた～」のような事態が発生します。

しかし、040turboならば、メモリ読み込みでなくとも、最後の「ミラクル・ロッマンスッ」のフレーズまで画像が停止することはなく（ほとんど一瞬で読み込み、展開が終わっているらしい）、５人全員のポーズを飛ばされることもなく、鑑賞することができます。この差は大きいですね（謎）。

さて、シャープの「インテリジェント・ビデオ・デジタイザ」も姿を見せはじめましたが、店頭デモを見るかぎり、ただのX68030では動きがガクガクしています。データの再生はMPUパワーに依存するようですから、これはやはり、040turboで試してみたいものです。

これで後は、大容量のSCSI２ディスクさえあれば、毎週「チャチャ」をSX-WINDOW上で録画・再生できるようになる（かも知れない）というものでしょう。

というわけで、040turboは、これからのCGA野郎の必需品となりそうです(思いっきり時節ネタですいません。^^;)。

（文●伊寿墨眼仁奈　NIFTY-Serve GHH01360）

アートワークがやってくる

あれやこれやといろいろ問題点を克服していくにつれ、バラック基板は結構安定して動くようになりました。原因不明の暴走がなくなり、一晩中動かしていてもコケなくなりました。

68030と比べて2ウェイト多く入ってしまうのは痛いところですが、安定動作が第一です。それに、キャッシュオンで動くようになればメモリアクセスの頻度が減るので、ウェイトのペナルティも軽減されるでしょう。

すでに、この時点ではHuman68kのコマンドだけでなく、ほとんどのアプリケーションが動くようになっていました。負荷が一番重いと思われるSX-WINDOWも問題なく動きますから、どこに出しても恥ずかしくないレベルです。もっとも、キャッシュオンにできないので、68030より遅いという情けない状態ですが。

そうこうしているうちに、基板のアートワークができあがってきました。業者から宅配便で送られてきた図面は、写真3.3のような、実際のアートワークのイメージをプロッタで印刷したものと、このアートワークをもとに、回路のどのピンとどのピンを結んでいるかを示す「ネットリスト」と呼ばれる情報を印刷したものです。回路図とネットリストを照合して正しく配線されていればOK。基板の製造に入るわけです。

途中、ちょくちょく細かい回路変更をしたので、ちゃんと反映されているかどうかも心配ですから丹念に照らし合わせていくと、残念ながら、3ヵ所ほど間違いが見つかりました。これではOKは出せませんので、修正をお願いする

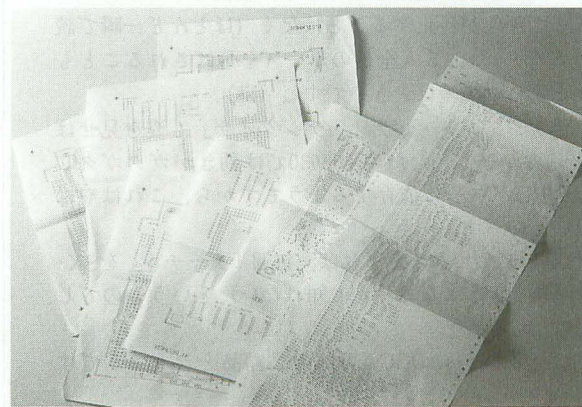



写真3.3　アートワーク確認用の図面

ことになります。

実際の修正作業自体はそんなに手間ではないのでしょうが、また確認のためにアートワークとネットリストを宅配便を使って送り返さなければいけません。最低でも2～3日はかかってしまう計算ですが、焦ってもしかたがありません。

さて、ネットリストが正しければ、基本的にはアートワークは正しいはずですが、最初のチェックですから、どの信号がどこを通っているかの確認もかねてアートワークの図面のほうでも1本ずつ信号線を赤ペンでたどってみます。クロックのパターンはどこを通っているかとか、変に回り道しているパターンはないかといったところがチェックのポイントです。*1

こうした作業を経て、やっと、アートワークにOKが出せました。

＊1
仕事で基板を作るときはここまでていねいには見ません。趣味のほうが気合いが入ります。

それでも、バグは隠れていた

基板配布を決意した時点ではバラック基板も動作が不安定だったので、たとえプリント基板化しても後からなんらかの改造が必要になるだろうと覚悟していました。しかし、アートワークができてくるまでの1ヵ月間に行ったハードウェアのバグ取りで、バラック基板は見違えるほど安定して動くようになっていました。

これなら、無改造でいけるかもしれない。

そう、ひそかに思いはじめていたのですが、やはり、そう甘くはありませんでした。アートワークのOKを出してそろそろ基板の製造が完了するという時期に、ひょっこりミスが見つかったのでした。

実はそれまでバラック基板で使っていたGALは16V8A-10LPと20V8A-15LPというものでした。10LPや15LPというのはそれぞれ遅延時間のことを表していて、10LPなら10nsec遅れるということです。これでも、名古屋のパーツ屋で売っているなかでは最も速いタイプでした。しかし、このシリーズには、7LPという、より速いタイプがあります。

X68030のクロックは25MHzですから1クロックの周期は40nsec、変換回路は立ち上がりと立ち下がりのそれぞれを使っていますから、実際には20nsec周期で動いていることになります。GALの遅延が10nsecや15nsecもあってはタイミングはアウトです。実際にはタイミングにマージンがあるので15nsec

タイプでも動きましたが、速いタイプが使えるなら、それにこしたことはありません。

最初に業者に渡した部品リストでは遅いGALでしたが、配布基板では最も速い７LPのタイプ[*1]を使うことにしました。

いちおう、７LPのタイプをバラック基板に載せてテストするつもりではいましたが、16V8B-7LPも20V8B-7LPも、手作りのGALライタでは焼けないBタイプです。市販のGALライタを通信販売で購入したり、GAL自体も名古屋には売っていないので、知人に頼んで秋葉原のパーツ屋で買ってきてもらったりと、いろいろ手間がかかってしまい、７LPでの動作確認が遅れていました。

＊1
このため、１枚あたりの単価が当初の見積もりよりもグッと上がってしまいました。

まあ、速くなってうれしいことはあっても、動かないことはないだろう。

と思っていたので、悠長に構えていたのですが、７LPのGALをバラック基板に載せて動かしてみて甘さを思い知らされました。

68040モードで起動してみます。問題なく立ち上がります。

ふむふむ、オーケーオーケー。

次に、SX-WINDOWを立ち上げます。GRWの中に一瞬ゴミが出ました。いやな予感がします。試しにpicファイルを表示させるとワサワサとゴミが表示されました。

なんだ、これは？

SX-WINDOWを終了し、画面モードをかえてグラフィック画面をオンにしてみます。

```
SCREEN 0,3, 1
```

目が点になってしまいました。

クリアされるはずの画面が、全面、砂の嵐です。グラフィックVRAMのアクセスでトラブって、クリアどころか、画面中にゴミをまき散らしていたのです。

１つずつ前のGALと交換していくと、20V8A-15LPのIC４は問題なく動作し、これを20V8B-7LPにするとゴミが出ることが判明しました。IC４は、ア

ドレスの下位４ビットおよびファンクションコードを生成しているGALです。

結局これは、IC４が受け持つアドレスの下位４ビットを変化させるタイミングが速すぎるのが原因でした。結局、この問題は、タイミングを半クロック遅らせることによって対処することができました。ただ、この変更には、GALのプログラム修正に加えて新たな２本の信号線追加が必要です。すでに基板は製造段階に入ってしまっていますから、アートワークの変更はまにあいません。

アートワークからやりなおすと、またまた大きな出費になりますから、できあがった基板に改造を施すことになりました。グラフィックといいテキストといい、X68030のVRAMアクセス*1には泣かされます。

それにしても、残念なのは７LPのGALの動作確認が遅れたことです。あと１週間早く試していたら、アートワーク作業に反映できたかもしれません。問題ないと思っていても、どこに落とし穴が潜んでいるかわからないものです。

しかし、この後、もっと大きなチョンボが発見されたのでした。

＊１
微妙なタイミングで動いているらしく、クロックアップしたマシンではほぼ確実にゴミが出ます。また、無改造のマシンでも、条件によってはVRAMにゴミが出ました。しかし、この対処はずーっと後になって行われました。

Voice of Users

ハンドメイドの楽しみ

はじめて040turboを手にしたときは感動しました。一言でいえば、

基板がきれい

これにつきます。さっそく家に持って帰り、X68030につけたら、あれ？　動かない？　どうしてだろう？　040turboの取扱説明書を見直しても大丈夫そうです。030モードも、040モードも駄目です。

もしかしたら、X68030が壊れたのか？

X68030を33MHzにクロックアップしているので、元の25MHzに戻してみましたが駄目。いろいろやっているうちに頭の中を購入代30万円がよぎり、「もうX68030は駄目か？」と思いながら、ふと040turboのボードを抜いて68030を挿してみました。すると、X68030は動くではないですか！　そのときは本当にホッとしました。その後、いろいろやっても駄目。

BEEPsさんに原因は何かと見てもらったら、なんとパターンカットの場所がちょっとずれていたらしく、別のパターンが断線していました。BEEPsさんに断線していた部分をつないでもらい、今ではちゃんと動いています。

私は、030モードと040モードを区別するLEDをどこにつけようかと迷ったあげく、ふと友達のXVIを見てみると、本体の色はX68030と同じだということに気づきました。XVIの前面パネルを買ってきてX68030の前面パネルと交換し、16MHz表示のLEDをPOWERに、10MHzの表示のLED*2を040モードにつなぎました。030モードと040モードの切り替えスイッチは、本体上面に穴を開けて取り付けているだけですが、あまりカッコよくないので、このへんもXVIのスイッチのようにしたいと思っています。

（文●SUPRA NIFTY-Serve MCN02045）

＊２
これは、さらに青色に取り換えました。

待ちに待った基板到着

さて、バグの発見によって基板無改造の夢が打ち砕かれ、ちょっとナーバスにはなっていましたが、それでも改造線2本の追加ですから簡単なものです。費用も見積もり価格の範囲内でやってくれるということで話がつきました。

そして待つこと1週間。1ヵ月半の時間を要して、やっと基板ができあがってきました。さすがに、いきなり第一次配布の人数分を作る勇気はないので、とりあえず評価用に3枚だけ作ってもらうことにします。この評価用基板に問題がなければ、すぐに第一次配布分の量産に入ってもらうというわけです。

9月も半ばの土曜日の夕方、待ちに待った評価基板が届きました。

まずは、じっくり眺めてみます。写真3.4.aができあがってきた基板です。さすがはプロの仕事です。美しい基板でした。しばし悦に入った後、さっそく改造線を張る作業に入ります。業者に改造してもらってもよかったのですが、少しでも早くほしかったので、改造に回さないで、そのまま送ってもらったのです。この改造は、バラック基板を作るのと比べれば手間のうちには入りません。

できあがった基板をさっそくX68030に取り付けます。X68030に長い間挿してあったバラック基板を外して、空いた68030のソケットに評価基板を取り付けます。写真3.4.bがその様子です。X68030を持ち込んで実装位置の検討をしただけのことはあって、ネジ穴の位置もピッタリ、まるで最初からそこにあったかのように、しっくりした出来映えです。

これが、プロの仕事ってやつなんだな。

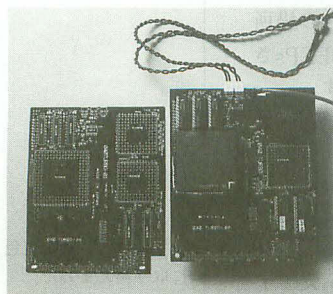
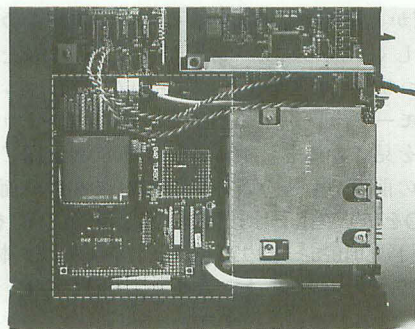



写真3.4　a.040turbo評価基板　プリント基板(左)　部品を載せた基板(右)

b.X68030に取り付けたところ

そんなことを思いながら、どこかショートしていないかを確認するため、テスターで電源とグランドを触ってみました。

天地がひっくりかえる大ショック

ピッピピーーーーーー

ショートを知らせるブザーです。抵抗値は、ほぼ０Ωを指していました。

何で？

どこかの部品どうしが接触しているのかと思い、あちこちからのぞいてみますが、それらしいところはありません。挿し込みミスで位置がずれていないかも確認してみましたが、間違いありません。

X68030のマザーボードから評価基板を外して、別々にテスターで当たってみると、評価基板もマザーボードも異状ありません。コンデンサの極性も確認してみましたが、どこにも間違いはありません。もう一度、慎重に挿しなおして再度テスターで当たってみると、

ピーーーーーー。

無情にも、またショートを知らせるブザーが鳴ります。

間違いなく、どこかおかしい。

それから、どれくらいの時間、あーでもないこーでもないと調べたことかわかりません。そして、やっと原因を突き止めたとき、全身の血の気が引きました。

マザーボードの68030のソケットと接続するために、評価基板側から突き出しているピンが、90度間違った向きになっていたのです。

写真3.5aがマザーボードの68030ソケットです。1番ピンは右上になっています。これに対し、写真3.5bができあがった基板の68030との連結部です。1番ピンが右下についているのです。

憂欝

すでに時間は土曜日の午後7時を回っていましたが、かまわず業者に電話をかけます。月曜日まで待っていられません。幸いなことに（向こうは不幸でしょうが）担当者は、まだ会社に残っていました。

「どうなってんだいこりゃあ」と熱くなる気持ちを抑えて事態を淡々と報告します。

す、る、と、

「そちらの部品レイアウトの指示どおりにやったはず」との返事。

ゲッ、またまた、血の気が引き、カチンカチンに凍りついていく感じです。

そういえば、確かにレイアウト案に1番ピンの位置も書いたぞ。
でも、どの向きで書いたか覚えがない。

調べてまた電話するからといってあわてて電話を切りました。控えはどこだったかとジタバタしていると、今度は向こうから電話がかかってきました。

「レイアウト案は正しかった、こちらのミス」とのこと。

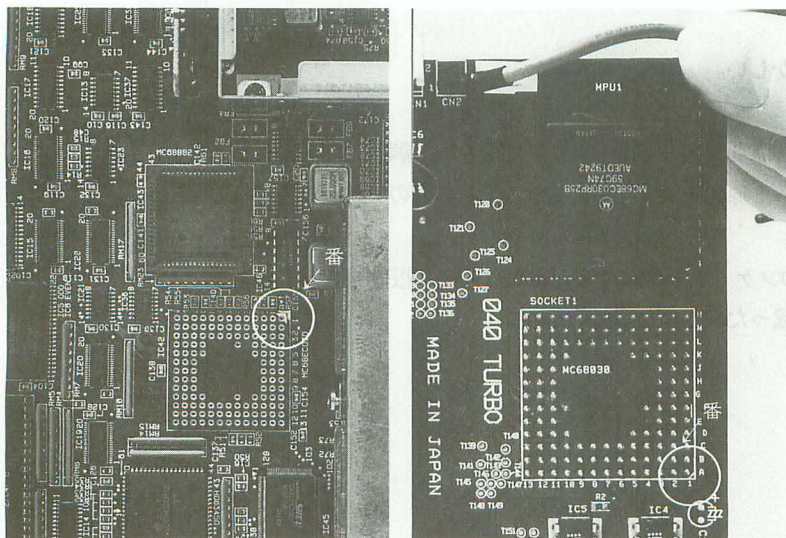
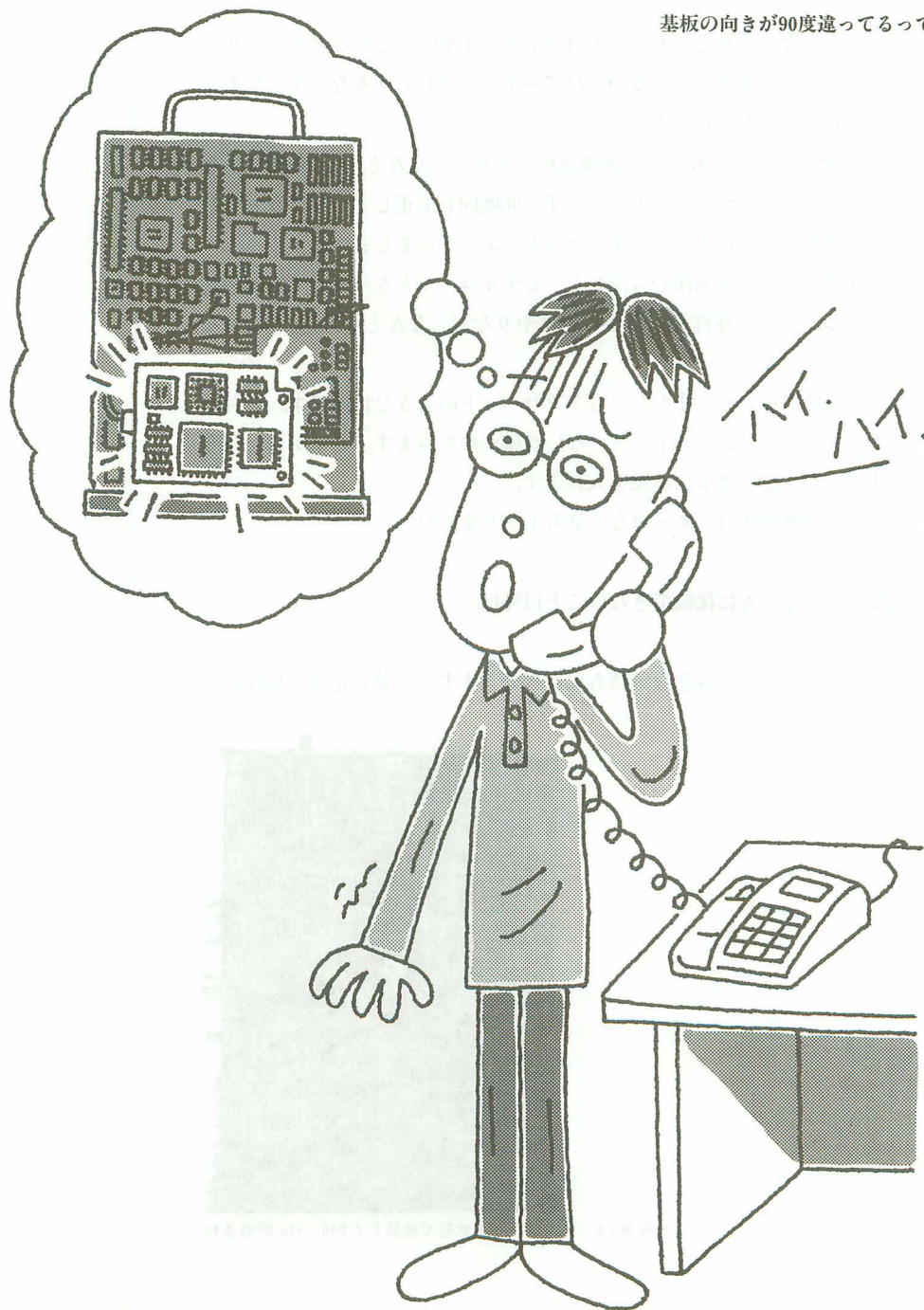



写真3.5　a.マザーボードの68030ソケット拡大(左)　b.評価基板の接続ピン拡大(右)

しかし、すでに凍りついていたので、ホッとするのが精一杯で、言われるままに「ハイ、ハイ」とうなずいて電話を切りました。

基板の向きが90度違ってるって!?

アクロバット接続の結果

普通のミスならなんとかなるかもしれませんが、128ピンもあるプロセッサの足の向きがごっそり間違っているのではアートワークからやりなおすしかありません。基板は作りなおしです。

しかし、せっかくできあがった評価基板ですから、なんとかX68030につないで評価だけでもしておきたいところです。90度回して正しくつなごうとすると、本体の基部と後ろにあるビデオユニットにぶつかってしまいます。

結局、マザーボードを本体から浮かし、ビデオユニットもバラして内部の基板だけにすることで、写真3.6のように無理やりながらなんとかX68030につなぐことができました。

バラック基板に続いて、またしてもアクロバットのような状態です。いちおう、電源とグランドがショートしていないか確かめてみます。さすがに、今度は大丈夫でした。恐る恐る、電源を入れます。

はじめての評価基板は、あっけなく立ち上がりました。

「完璧」……まともに接続できないこと以外は。

そう思うと、よけいに残念な気持ちになってきます。一発で完璧な基板がで

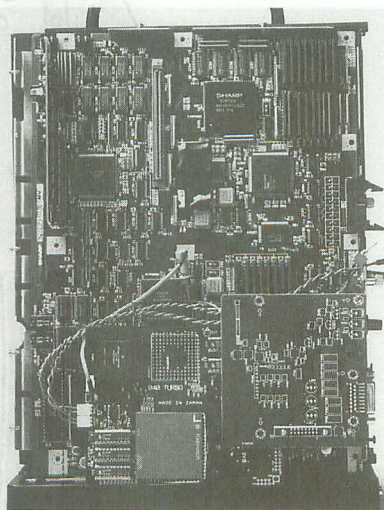



写真3.6　アクロバット状態で接続した040turbo評価基板

きるとは思っていませんでしたが、こんなミスでアートワークのやりなおしになるとは予想もしていませんでした。

このアートワークのやりなおしはさすがに業者側が費用を全部持つことになりましたが、これで配布も当初の予定より大きく遅れてしまいました。しかし、こちらもアートワークの図面チェックで見落としをしていたわけですから、一方的に相手のミスを責めるわけにもいきません。それに、GALを７LPの高速タイプに変更するための回路修正も、ここでいっしょにしてもらうことができました。怪我の功名というには、ちょっと痛すぎるケガでしたが、しかたがありません。

Voice of Users

苦あれば楽あり　040turboライフのすすめ

このボードのことは以前からネット上で話題に上っており、ぜひ私も使用してみたいと思っていたとき、NIFTY-Serveに入会している友人から配付の話があると聞き、ふたつ返事で参加させてもらいました。

しかし、最初はなんとも安定しない状態でした。040モードではゲーム「悪魔城ドラキュラ」も走ってしまうほど安定動作していたのですが、030モードでは電源を入れて１分ほどするとコケるのです。

そのうちメインボードのIC９にコンデンサをかませることによって回避できると聞き、さっそくやってみたところ、この症状は出なくなりました。ほとんどの方はこのような症状は起こらないそうで、X68030の個体差によるものだったようです。

これでやっと安心して040turboを付けることができました。

はじめてコピーバックモードで動作させたときは本当に感動ものでした。

たとえば、LHAによる約1300Kバイトほどのファイルの圧縮は、10MHz機種では遅くてやる気にもなりませんでしたが、040のコピーバックモード（25MHz）では約１分と信じられないほどの速度です。TeXのコンパイルもWSに匹敵するほどですし、JPEG展開は平均すると約３秒です。今まで「68は遅い遅い」といわれていましたが、これなら遅いと感じることはないはずです(しかし、最近では贅沢なことに、これでも遅いと感じるようになってきてしまいました。従来機種のみなさんに怒られそうですが。^^；)

また、速度があまり必要ではないときでも040モードでコケることはまずないので、私は大体040モードで立ち上げています。まあ、ゲーム等をするときは030モードで十分ですから、必要に応じて２つのモードを切り替えて使えるのが、このボードの優れたところですね。

このように、私のパソコンライフの中に完全に040turboは溶け込んでいます。もう手放せないといった状態です。

（文●ごーどん　MAX-BBS MAX0230）

第 4 章

68040ソフトウェア

いよいよソフトウェアに取りかかる

当初の予定では、第一次配布でみんなの協力を得ながらソフトウェア側の問題に当たっていくつもりでしたが、基板ができないのではしかたがありません。アクロバット状態とはいえ、評価基板は見事に安定動作してくれましたので、ソフトウェアのほうに手をつけることにしました。

今までは重要な作業をするときは万一のことを考えていちいちバラック基板を外し、68030に挿しなおしていましたが、この基板なら、そのまま作業することができそうです。本格的に68040をキャッシュオンで動かすためのプログラム製作に取りかかります。

さて、68040をキャッシュオンで動かすためには、次の点が問題となります。

1）キャッシュの制御方法の違い
2）非シリアライズドアクセスの問題
3）動作速度の違い
4）バーストアクセスの問題

これらの問題、実は最初からわかっていたわけではありません。68040のユーザーズマニュアルを見ていて、なんとなく問題がありそうだなあとは感じていましたが、ちゃんとは考えていませんでした。

実際にハードウェアがトラブルなく動くようになってはじめて、

さあ、キャッシュオンにするぞ。

と真面目に考えはじめたようなものです。

それでは、それぞれについて、順に説明していきましょう。

1）キャッシュの制御方法の違い

前にもちょっと触れましたが、68030と68040はキャッシュの制御方法が違っているため、68030のキャッシュ制御命令を使ったプログラムのままでは68040のキャッシュを制御できません。

ちなみに、X68030の場合、IOCSルーチンにプロセッサのキャッシュ制御を行うコールIOCS-＄AC[*1]が追加されました。ほかのプログラムがすべてこ

＊1
XC ver2.1 NEW KITでSYS_STATという名称になっていました。この時点では知らなかったので、「キャッシュ制御のIOCSコール」、もしくは単に「IOCS-$AC」と呼んでいました。今でも040turboのドキュメント類はそうなっています。

のIOCSコールを使って制御するような作りになっていれば、この部分だけを68040用に書き換えればすむのですが、残念ながら、そう甘くはありません。

ROMのIPLルーチンはもとより、他のIOCSコールさえも、このルーチンを使わずにキャッシュを制御しています。調べてみると、Human68kも直接68030のキャッシュ制御命令を使っています。このため、キャッシュ制御命令*1を使っている部分を洗い出して片っ端からパッチしていかなければなりません。

＊1
68030はキャッシュ制御レジスタを使ってキャッシュを制御するため、制御レジスタをアクセスするmovec命令を使用することになります。この命令は、機械語コードで＄4E7Aおよび＄4E7Bになるので、このコードをサーチして、これを手がかりに前後のプログラムを解析すれば、比較的簡単にチェックすることができます。

2）非シリアライズドアクセスの問題

シリアライズドアクセスとは、命令の並び順にデータをアクセスするよう68040に強制するモードです。常識的に考えると、これが普通の動作のような気がしますが、性能重視の68040は、デフォルトが非シリアライズドアクセスモードになっています。

68040は、メモリの先読み機構と独立した書き込み機構があり、それぞれ命令の実行部とは独立して動くようになっています。このため、メモリへの書き込み命令の次に読み込み命令があったとしても、必ずしも書き込み、読み込みの順になるとはかぎりません。

もちろん、同じアドレスへの書き込みと読み込みでしたら、ちゃんと順番どおりになるようにロックされますが、別のアドレスであればズンズンおかまいなしに進んでしまいます。しかし、I/Oアクセスなどでは命令の順番どおりにならないと困る場面もあるでしょう。このため、そういうときはシリアライズドアクセスモードにしなければなりません。

ちなみに、I/Oをアクセスしてキャッシュに載ってしまうと、次にはキャッシュ上のデータを読んでしまって実際のI/Oポートの変化がわからなくなるという心配もありましたが、調べてみたところ、X68030は16ビットサイズのデバイスへのアクセスについて、つねにCIIN（Cache Inhibit IN）信号を返すようになっていました。このため、I/Oポートをアクセスして得られるデータはキャッシュには載らず、つねに実際のアクセスが行われますので、心配はいりませんでした。

しかし、I/Oだけでなく、VRAMやI/Oスロットに挿したメモリボードのアクセスも、すべてキャッシュ禁止にされてしまうのはちょっと考えものです。

話が脇道にそれましたが、とにかく、シリアライズドアクセスモードにしないとまずいだろうと思っていました。でも、実際にアクセスの状況を調べてみると、あまりアクセスの逆転現象は起こっていませんでしたし、デフォルトの非シリアライズドアクセス状態のままでも別に問題が起こりませんでしたので、気にする必要はないのかもしれません。

3）動作速度の違い

これは、68040に切り替えると命令の実行時間が短くなるために、プロセッサの実行速度に依存したプログラムで不具合が出るというものです。

I/Oのなかには、アクセスに一定時間の余裕を持たせないと誤動作するものがあります。キャッシュオンで動作させると命令フェッチが入らなくなりますから、連続してアクセスできてしまう場合も起こります。nop命令[*1]の数やソフトウェアループでタイミングをとるようなプログラムですと、プロセッサの実行速度の違いのため、目的の時間より短くなって不具合が出るでしょう。

実際、ループを回るだけのpv.xで、68040は初代比で16倍、68030比でも3倍以上ですから、ループでタイミングをとると所要の時間をかせげません。

X68030のROMルーチンは、どういった対策をしているのだろうかと解析してみると、次のようなプログラムが見つかりました。

```
1:  TST.B    $E9A001
2:  TST.B    $E9A001
3:  MOVE.B   #$08, $E8A01B
```

3行目の$E8A01B番地はRTC（Real Time Clock）のレジスタの1つ[*2]です。このアクセスの前に、$E9A001番地を2回テスト命令[*3]でアクセスしていますが、その結果は使われていません。単に読み捨てているだけです。この$E9A001番地は何のI/Oかと調べてみると、ジョイスティックポート[*4]でした。RTCと本来何の関係もないジョイスティックポートですが、これらのI/O系は10MHz[*5]で動作していますから、アクセスすれば確実にウェイトが入ります。プロセッサの実行速度に依存するnop命令でタイミングをとるよりも確かに効果的ですので、何回アクセスしても以後の動作に影響を及ぼさないジョイスティックポートをアクセスすることでウェイトをとっているのです。

実際、この点に関しては特に対処はしませんでしたが、問題はありませんでした。

X68000時代のフリーソフトウェアのなかにはキーリピートが速くなるといった問題が出るものもありますが、そういうプログラムはたいてい68040でなくても、X68030で動かす時点で問題が解決されていますから、特に考えないことにします。

4）バーストアクセスの問題

実は、この問題については第一次配布の直前まで気がつかなかったので、おおいに悩まされました。

＊1
68000では単にノーオペレーションでしたが、68030や68040では直前の命令の実行の完了を待つという機能も持っています。ちなみに、他の命令の場合は、前の命令にオーバーラップしてどんどん実行されていきます。

＊2
RTCのモード設定レジスタ。

＊3
TST命令は、値が〝0〞かどうかをテストする命令です。

＊4
8255のAポート。

＊5
実際には12.5MHzのようです。

TBI（Transfer Burst Inhibit）信号をつねにアサートしているし、68030のCIIN信号を68040のTCI（Transfer Cache Inhibit）信号につないでいるので、68040のキャッシュも68030とはサイズの違いこそあれ、同じように扱えると思っていたのです。

68030も68040も、キャッシュは4ロングワードを1ラインとして管理されていますので、1バイトといった半端なデータをアクセスする命令を実行しても1バイトだけ読み込んでキャッシュに置くといったことはしません。

しかし、68030はバーストモードを設定しないかぎり、デフォルトでは1ロングワード分アクセスしてキャッシュに置くだけ[*1]なのに対し、68040ではつねに4ロングワードを一気にアクセスしてキャッシュの1ライン分を丸ごと充塡しようとします。

プログラムで使ういろいろなデータ領域は、普通、ある程度近いところにまとめて置かれますから、1ライン分あらかじめアクセスしてしまっても無駄にはなりません。それに、68040はバースト転送を前提にしていますので、1ライン分を一気にアクセスしたほうが得なのです。

実は、このなんでも1ライン単位でアクセスするという仕様が曲者だったのです。たとえTBI信号をアサートしてメモリ側がバースト転送できないと68040に教えても、普通のロングワード転送には変わりますが、やはり、4ロングワード分アクセスして1ラインを充塡[*2]しようとすることに変わりありません。

また、TCI信号をアサートして、アクセスしたデータをキャッシュしてはいけないと教えても、やはり、4ロングワード分のアクセス[*3]をしてくるのです。

これは、I/Oポートのアクセスに致命的な問題を引き起こします。

あるI/Oポートをアクセスすると、プログラマが知らない間に付近の4ロングワード分のI/Oポートまでもが勝手にアクセスされてしまうのです。これを回避するためには、I/Oポートアクセスをキャッシュオフで行わなければなりません。これは、先の非シリアライズドアクセスモード同様、プログラマが予期しない動作をするのですから、いくら性能のためとはいえ、やっかいな問題といえるでしょう。

＊1
「シングルエントリモード」と呼ばれます。ロングワード境界にまたがるデータの場合は、2ロングワードアクセスします。

＊2
もちろん、データキャッシュがオフなら、こんなことはしません。

＊3
結局、このデータはキャッシュに残されませんので、無駄になります。

システムソフトウェアへのパッチ

68000は最初から32ビット指向のプロセッサだったので、68000シリーズのプロセッサは互換性が高いといわれていましたが、68040はハードウェアの性能を重視して設計されているため、システムレベルでは結構いろいろな問題があることがわかったかと思います。

もっとも、68030を前提としたハードウェアとソフトウェアに個人で勝手に68040を載せようとしている*1こと自体に無理があるのですから、文句はいえません。

しかし、肝心のOS*2のほうが68030ベッタリで書かれていますから、68040用にパッチしなければなりません。

アプリケーションプログラムに関しては、後で個別にパッチを用意するとして、まずはシステムレベルのプログラムに対するパッチを考えてみます。SX-WINDOWもシステムレベルのプログラムといえますが、Human68kの起動後に実行しますから、一種のアプリケーションとみなして、これも後回しにします。

すると、最低限パッチで必要なプログラムとしてはROMルーチンとHuman68k、それにフォーマット時にハードディスク上に書き込まれるSCSIのデバイスドライバ（SCHDISK）にターゲットが絞られました。さっそく、これらに対してパッチを行うプログラムの開発に取りかかります。

＊1
キャッシュ制御やI/Oアクセスの問題などにユーザーはタッチしないというのが、お約束です。こういう部分の違いはOSが吸収するのであり、OSはベンダーが提供するものだから、ユーザーは気にしないでよいというのが、モトローラの論理です。

＊2
Human68kを捨てて、68040用に別のOSを使用するという、清く正しい道もあります。NetBSDなら申し分ないでしょう。

ROMルーチンへのパッチ

さて、ROMルーチンへのパッチといっても、ROM自体を書き換えることはできません。X68030のROMは、起動時にプロセッサの判別を行うようになっており、68010や68020にも対応していますが、残念ながら、68040には対応していません。

68040対応を加えたROMを配るという手もありますが、費用もバカになりませんし、第一、著作権の問題があります。キャッシュオフ状態なら、ノーパッチの状態でも問題なくHuman68kが起動しますので、なんとかプログラムで対処する方法を考えてみます。

IOCSコールはRAM上にエントリポイントがあるので、ROMのキャッシュ制御部分のかわりにRAM上に修正したルーチンを用意し、ここに飛んでくるようにエントリポイントを差し替えることで対処することができます。しか

し、実際には、多くのIOCSコールがROM内のキャッシュ制御ルーチンを直接コールしているため、この方法は修正箇所が多くなって面倒です。そこで、いったんROMルーチン全体をRAM上にコピーしたうえで問題箇所だけを修正する方法をとりました。

そして、IOCSコールのエントリポイントや各種割り込みベクタに書かれている飛び先アドレスが、このRAM上のルーチンになるようにするために、RAM上で修正をかけた後、このRAM上に持ってきたIPLルーチンで再起動させるようにしました。こうすると、ほとんどのルーチンが、このRAM上のルーチンをコールするように初期化されます。

ただし、この方法の欠点は、RAM上にROMのコピーを置くため64Kバイトほどメインメモリを消費してしまうことです。とはいっても、メモリ制限のキツイMS-DOSマシンじゃあるまいし、64Kバイトなんてゴミみたいなものですから、ここは大目に見てもらうことにします。ROMの絶対番地を意識するプログラムだと問題になりますが、そういう、お行儀の悪いプログラムについては考えないことにします。

さて、ROMルーチンへのパッチのめどがたったところで、この方法でパッチするプログラムを作ってみました。パラメータとして、ROMのコピーを置くアドレスを指定して実行します。

```
040ROMpatch /BF0000
```

といった感じで実行すると、ROMの$FF0000～$FFFFFFの内容を$BF0000～$BFFFFFにコピーし、パッチしたうえで、ここから再起動します。立ち上がった後は、ROMのパッチができているというわけです。

なお、ROMのコピーを置いた$BF0000～$BFFFFFが、Human68kのプログラムで壊されないようにするために、フリーエリアを制限しておきます。この場合であれば、switch.xでメモリを11Mバイトに制限することで、$B00000以降の1MバイトがHuman68kの管理外になるので安全です。

Human68kへのパッチ

Human68kへのパッチは簡単です。

Human68kのプログラムはRAM[*1]ですから、問題箇所に直接パッチを当てればよいのです。フロッピーディスクやハードディスク上のファイルとして存在するHuman.sysを直接書き換えてしまってもよいのですが、そうすると、68030モードと68040モードで起動方法を変えなければならなくなるので面倒

＊1　スーパーバイザ領域としてハードウェア的に保護されていますので、ユーザーモードから書き変えることはできませんが、スーパーバイザモードになってしまえば書き換えできます。

です。68040でもキャッシュオフならノーパッチで起動するので、こういった実行時のパッチという芸当ができるのです。

これについても、パッチプログラムを作ってみました。

```
040HUMANpatch.x
```

といった感じで実行します。

ただし、単独で実行しても意味がありません。まず、先の040ROMpatch.xを実行してROMルーチンのパッチを当てて再起動が行われた後、040HUMANpatch.xを実行すると、めでたくキャッシュオンにできるのです。

しかし、まだこの時点ではパッチが完全でなかったために命令キャッシュをオンにすることはできましたが、データキャッシュをオンにしてフロッピーをアクセスすると暴走してしまいました。

I/Oアクセスが悪いらしいということまではわかったのですが、原因が突き止められなかったため、MMU（Memory Management Unit）の付属機能である透過制御レジスタを使って、スーパーバイザモードではデータキャッシュがオフになるように設定しました。

MMU自体を使っていないのでアドレス変換もなにもないのですが、透過制御レジスタは命令メモリ、データメモリのおのおのについて、

・スーパーバイザモード／ユーザーモードの区別
・キャッシュモードの区別

C O L U M N

MMUと透過制御レジスタ

MMUは、プログラムに書かれたアドレス（論理アドレス）と、実際にアクセスされるアドレス（物理アドレス）を付け替えるメカニズムですが、この透過制御レジスタに設定される領域に関してはこの変換が行われずに、プログラムに書かれた論理アドレスがそのまま物理アドレスとなって出力されます。これを称して、「透過」というわけです。

この透過制御レジスタには、命令用、データ用にそれぞれ2本ずつ、計4本用意されており、ROM領域や、VRAMエリアなど、MMUでアドレス変換する意味がない固定領域をアクセスする目的で用意されています。

なお、透過制御レジスタは、アドレスの上位8ビットを指定して、それに該当するアクセスを透過領域としますので、指定できる領域は16Mバイト単位になります。

が独立して設定ができるのです。

そこで、データ用の透過制御レジスタに対し、

・スーパーバイザモード
・キャッシュオフ

という設定をしておけば、たとえキャッシュオン状態であっても、スーパーバイザモードでのデータアクセスに関しては、透過制御レジスタの設定が優先されて、キャッシュオフとしてアクセスされるのです。

これで、いちおう、I/Oアクセスの不具合がなくなり、データキャッシュもオンにできるようになりました。

しかし、透過制御レジスタの設定単位が16Mバイトと大きいため、I/O領域のみならず、X68030のメモリ空間を覆ってしまい、スーパーバイザモードで動くプログラムが通常のメモリをアクセスするときもデータキャッシュがオフになってしまいます。スピードを求めてスーパーバイザモードに移行して突っ走ろうとするプログラムほど遅くなるという皮肉な結果になってしまいました。

SCHDISKへのパッチ

SCHDISKは、SCSIハードディスク上に書かれたデバイスドライバです。format.xがハードディスクをフォーマットするときにIPLといっしょに書き込みます。

実はSCSIには苦手意識があって、SCHDISKに関してはしっかり解析していないうえに、ちょっと不思議なことがあって、このパッチについてはあまり自信がありません。

というのも、最初の頃はハードディスクを壊すのを恐れてフロッピーベースで運用していたせいもあって、SCHDISKの内部にキャッシュ制御コードが入っていることに気づきませんでした。そのため、かなりの期間、SCHDISKはノーパッチ状態で使われていたのですが、特に問題は出なかったのです。後になってメモリ上でキャッシュ制御命令の「movec」が残ってないか探していたら、SCHDISKのなかに「movec」を発見して、あわててパッチプログラムに追加したのです。

しかし、考えてみると、X68030登場前の、Human68k ver 2の時代のformat.xでフォーマットしたSCSIハードディスクに入っているSCHDISKは、当然、キャッシュについて考慮されていません。それなのに、こういったハードディスクをつないでもなんら問題が起こっていませんので、Human68k側やIOCS

ルーチン側でガードがかかっているのではないかと考えられます。

調べてみると、確かにデバイスドライバを呼ぶときはわざわざデータキャッシュをオフにしていることがわかりました。SCHDISK内にあるキャッシュ制御命令は、念のため、入れてあるだけにすぎないのかもしれません。いずれにしても、パッチしておくにこしたことはないので、このパッチデータも追加することにします。

040SYSpatch.sys

ROMルーチンとHuman68kへのパッチによるキャッシュオンの動作が確認できたので、このパッチデータをもとにシステムに対して一括してパッチを行うデバイスドライバを作ることにしました。入出力をするわけではないので、本来の意味のデバイスドライバではないのですが、X68000にはシステムに対する機能拡張を目的として具体的な入出力機能を持たないデバイスドライバも多数ありますから、それらと同じです。

さらに、デバイスドライバにすることにはもう1つの意味があります。

040ROMpatch.xでは、メモリサイズを制限してROMのコピー領域に割り当てていましたが、スマートではありません。勝手にシステムから必要なメモリを確保してくれたほうがきれいです。

しかし、Human68kが起動した後だと、図4.1のように、RAMディスクやディスクキャッシュがメモリに陣取ってしまうため、ROMのコピー領域を確保しようとしても中途半端な位置にきてしまいます。パッチ方法を手抜きしたせいもあるのですが、元のROMが$FF0000から始まっている[*1]ので、ROMのコピーを置く領域は下位16ビットがオール0になる、$BF0000といった、きりのいい番地からにしないと都合が悪いのです。

そこで、最初に起動して、RAMディスクドライバなどメモリの後尾からバッファを確保するプログラムが走り出す前に、メモリの最後尾を押さえてしまおうというわけです。プロセッサの問題に対処するためのパッチという意味からいっても、最初に起動するのは望ましいでしょう。このためには、単純に、config.sysの最初のdevice文でパッチプログラムが実行されるよう、デバイスドライバの形態にする必要があります。

ちなみに、Human68kにはprogram文という、config.sys内に記述することで通常のプログラムを実行させる機能もありますが、これは、たとえdevice

＊1
パッチが必要なIPLやIOCSが置かれた領域がここからということで、ROM自体はもっと前のアドレスからあります。

文より前に書いておいても、実際の実行はdevice文によるデバイスドライバの登録後になってしまうようなので、今回の目的には使えません。

パッチプログラムのデバイスドライバ化

デバイスドライバにするとデバッグが面倒になるのですが、パッチデータについては、いちおう、040ROMpatch.xと040HUMANpatch.xで動作を確認しているので問題はありません。まずは、これらのプログラムを、単にデバイスドライバの形態に置き換えました。デバイスドライバの形態といっても難しいものではありません。それに、今回のプログラムは入出力も何もなく、ただ単にデバイスドライバの形態で最初に１回動けばいいだけですから、置き換え作業自体は単純です。

ここで、１つ、失敗談を紹介しておきましょう。

デバイスドライバにするファイルには、デバイスヘッダという情報を先頭につけることになっており、このなかで指定するデバイス名という情報によって、そのデバイスのファイル名が決まります。どうせ入出力はしないから、デバイス名はなんでもいいだろう思い、縁起もので〝040TURBO〟という文字列にしたのですが、これが大ハマリでした。〝040TURBO〟という名前のファイル名が使えなくなってしまう＊[1]のです。

しかし、やっていることがなにせHuman68kへのパッチ当てです。〝040TURBO〟というディレクトリやファイルアクセスでエラーをくらうようになるものですから、とうとうディスクを壊したかと焦りました。今はデバイス名の頭の１文字目を$01にしています。頭にコントロールコードがくるようなファイル名は普通使いませんから、困ることはありません。

さて、デバイスヘッダはおまじないのようなものですから別に問題ないとして、プログラム自体にも新たに処理を追加しなければなりませんでした。１つ

＊１
よくopmdrv.xがないときに、うっかり〝opm〟というファイルを作ってしまい、opmdrv.xを登録した後で消せなくなるのと、原理は同じです。

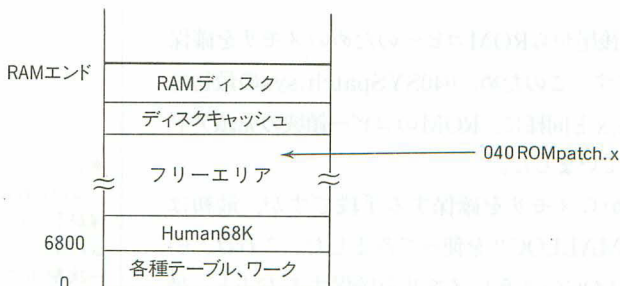



図4.1　Human68k起動後のメモリマップ

は、ROMにパッチが当たっているかどうかを判断する処理です。

040ROMpatch.xを使っているときは、まず、ROMにパッチを当てて、再起動したら、040HUMANpatch.xという具合に人間が順番に実行すればいいのですから、パッチが当たっているかどうかなど判断する必要はありませんでした。

しかし、040SYSpatch.sysは、config.sysに追加して、起動時に自動的に登録されるようになります。このため、ほうっておくと、ROMにパッチを当てて再起動、また、040SYSpatch.sysが走り出して再起動、というふうに無限に再起動を繰り返してしまいます。割り込みベクタやIOCSのエントリがROMを指しているかどうかを調べるだけでいいことですが、こんな簡単なことも結構気がつかないものです。

[SHIFT]キーを押しながら起動するとバッファをクリアするRAMディスクをまねて、最初は特定のキーを押したときだけ再起動するようにしていました。

同じような話で、68030のシステムではこのパッチは不要ですから、これまた、別のキーを押しているときだけスキップするようにするとかしていました。これも、動作しているプロセッサが68040か68030かをチェック*1すればいいだけのことです。

まあ、それでも、なんとかデバイスドライバの形態で登録し、起動時にシステムに対してパッチを当てるようにできました。これで、いちいちパッチコマンドを順番に実行していたときよりもずっと使い勝手がよくなりました。

*1
68030と68040の切り替えがソフトウェア的に行えるようになっていれば、特定のキーを押しながら立ち上げると68040モードになるということも可能になり、カッコイイのですが、今回は変換回路の簡略化のため、機械的なスイッチの切り替えになっています。

メモリ確保

デバイスドライバの形態にしたのは、システムから適当にメモリを確保するように作りかえるのが目的の1つでしたが、この時点では、まだ目的は達成されていませんでした。

というのも、メインメモリの最後尾からROMコピーのためのメモリを確保する手段がわからなかったためです。このため、040SYSpatch.sysの最初のバージョンでは、040ROMpatch.xと同様に、ROMのコピー領域の先頭アドレスをパラメータ*2として指定していました。

さて、メインメモリの最後尾からメモリを確保する手段ですが、最初はHuman68kのDOSコールにあるMALLOC 2を使ってみました。これは、引き数の指定によって通常のMALLOCのようにメモリを確保する方法と、最後尾から確保する方法のどちらかを選択できるものです。

しかし、うまくいきません。

*2
この仕様はメモリ確保がうまくいかないときの保険の意味で、第一次配布に添付した040SYSpatch.sysにも引き継がれましたが、今ではなくなりました。

その後に登録したRAMディスクドライバは、あいかわらずメモリの最後尾からバッファを確保しています。常駐プログラムではMALLOC 2で確保した領域はそのまま解放されずに残るようですが、どうもデバイスドライバの場合は違うようです。しかし、メモリの後尾からバッファ領域を確保するデバイスドライバはちゃんとバッファ領域がかちあわないでうまくいっていますので、何か、ほかにうまいやり方があるのでしょう。

しかたがないので、純正のRAMディスクドライバ[*1]を逆アセンブルして、どんな方法を取っているのか調べてみました。

その結果わかったことは、$1C00番地の値を直接操作するという方法でした。ここには、最初にフリーエリアの最後の番地が書き込まれており、RAMディスクドライバなど、バッファを確保したいプログラムは、この$1C00番地のデータを直接操作していました。図4.2.aのように、040SYSpatch.sysが最初にコピー領域として必要なメモリ容量を引いて、残りの値を$1C00にセットしなおします。次のプログラムがメモリの後尾からバッファを確保しようとするときは、あらかじめ確保したいプログラムが$1C00番地の値を基準にしますので、図4.2.bのようになり、バッファ領域がかちあうことはありません。Human68kのワークを直接書き換えるという荒業ですが、純正のRAMディスクドライバがやっている手法ですから、これに見習うことにします。

これでやっと、040SYSpatch.sysも自分でメモリを確保することができるようになり、その後に登録したRAMディスクがROMのコピー領域をつぶすこともなくなりました。

＊1
〝RAMDISK.SYS〟という純正RAMディスクドライバです。X68000の頃からついているのはいいのですが、お世辞にも速いとはいえません。下手をすると、SCSIのハードディスクより遅いという代物です。私も普段は、フリーソフトウェアのRAMディスクドライバを使っています。

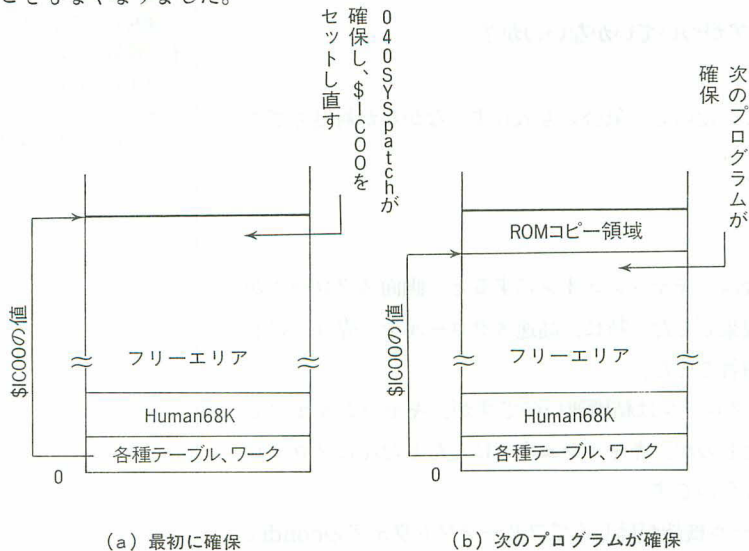



図4.2　フリーエリアの後尾からメモリを確保する方法

まだ残る不具合

最初の頃は、透過制御レジスタを使ってスーパーバイザモードでのデータキャッシュを禁止していましたが、FDC（Floppy Disk Controller）からの割り込みルーチン内にあったキャッシュ制御ルーチンにもパッチを当てることで、いちおう、データキャッシュオンでフロッピーディスクをアクセスしてもハングアップ[*1]しなくなりました。

こうして、まずそうな箇所をパッチすることで、いちおう、68040もフルタイム、キャッシュオンで動かせるようになりました。しかし、実際に使用してみると、まだいくつかの不具合がありました。

現象からいうと、次の３点でした。

1）スクロールが遅くなる
2）マウスカーソルを表示するとゴミが出る
3）see.x[*2]でフロッピーをアクセスするとハングアップする

調べてみましたが、なかなか原因がつかめません。壁にぶち当たった感じです。

68040の速度にハードウェアがついていかないのか？

そんなことを思いはじめていたので、気合いも入らず、なかなか対処もできませんでした。

1）スクロールが遅くなる

最もはっきりわかる不具合は、キャッシュオンにすると、画面スクロールが目に見えて遅くなるという現象でした。特に、高速スクロールを〝売り〟にしているアプリケーションで顕著でした。

もともとX68030の画面スクロールは結構速いのですが、キャッシュオフでは軽快にスクロールしていたものが、キャッシュオンにしたとたんにノロノロしたスクロールになってしまうのです。

私は、普段、バックスクロール機能がほしくてフリーソフトウェアのcondrv.sysを使用していましたので、最初はこれが悪いのだろうと思って気にしてい

＊1
実はまだ本質的な解決になっておらず、最終的には第一次配布の後まで問題を引きずりました。

＊2
フリーソフトウェアのファイルビューワプログラム。lhaのアーカイブファイルの中身も、いちいち展開せずに覗くことができるので便利です。たまたま私が愛用していたのでこれを引き合いに出していますが、see.xが悪いわけではありません。

ませんでした。実際、condrv.sysはX68030登場前からあったため、X68030の登場当初はいろいろと不具合があったのです。この不具合を回避するため、暫定的なパッチ情報が公開されて、やっと使えるようになったという経緯があります。今回も、68040とcondrv.sysとの相性の不一致だろうと思い、アプリケーションのパッチは後回しにしていましたから、スクロールが遅くなっても気にしなかったのです。

ところが、あるとき、condrv.sysも含め普段使っているデバイスドライバや常駐ソフトをいっさい外してHuman68kと040SYSpatch.sysだけにして使ったところ、やはりキャッシュオンするとスクロールが目に見えて遅くなることに気づきました。X68030登場時、高速スクロールするプログラムがラスタスクロールでゴミをまき散らすという問題があったので、これについても、

68040になって処理スピードが上がり、高速スクロールのために微妙にチューンナップされたプログラムのタイミングが変わったのかな？

と思っていました。しかし、結局、この問題は第一次配布の後までもつれこみました。

C O L U M N

040turbo第一次配布

第一次配布の募集をしたときは、あまり安定して動いていなかったこともあって、デバッグに参加してもらうスタッフ募集という位置付けでしたので、私が連絡のとれる範囲としてNIFTY-ServeおよびMAX-BBSに案内を出しただけでした。

誰も参加してくれなかったらどうしよう。

と心配していましたが、最終的に40人近くの参加者が集まってくれました。
しかし、68040自体の値段が高かったことや、部品代が見積もり価格より上回ったこともあって、結構な値段になってしまい、

これで動かなかったら冗談ではすまされないなあ。

とビビッていました。第一次配布の後のトラブル続き、バグ続きのときは冷や汗ものでした。辛抱強くつき合ってくれた第一次配布参加者の方々に感謝しています。
今は笑って話せるようになって、本当によかったと思っています。

2）マウスカーソルを表示するとゴミが出る

これもまた、やっかいな問題でした。X68030は、マウスの右ボタンを押すとマウスカーソルが表示されます。そして、040SYSpatch.sysの場合、マウスカーソルが表示されると、画面の関係ないところにゴミが出るのです。

しかも、マウスを動かすと、写真4.1のようにゴミが広がっていくのです。

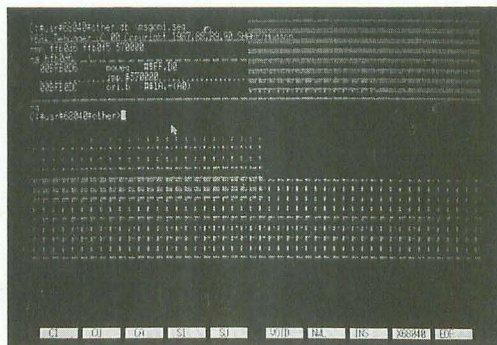



写真4.1　マウスを動かすと出るゴミ

ソフトウェアキーボードを表示させようものなら、画面中ゴミだらけになってしまいます。もともと、グラフィックVRAMのアクセスでバスエラーが出るなど、VRAMアクセスまわりはどうもタイミングが怪しいのです。

またハードウェアの問題か？

ロジックアナライザで調べてみたのですが、変なデータを書き込んでいるような形跡はありません。それに、キャッシュオフならゴミは出ません。画面モードを切り替えてグラフィック画面を表示状態にすると、ゴミが出なくなったりすることがあったため、原因が絞り込めずに混乱してしまいました。

とにかく、マウスカーソルを表示させるとはっきりゴミが出ていたのが、何かをいじるとピタリと止まります。これで直ったと喜んで、電源をいったん切って再度試してみると、やっぱり出るというわけです。いろいろいじってみたのですが、結局、ハードウェアのほうでは対処できませんでした。

つかみどころのない現象でほとほと困っていたのですが、よく整理して考えてみると、マウスの青色を表示しているプレーンにだけゴミが出て他のプレーンにはまったくゴミは出ていないという点に何かカギがありそうだと気づきました。テキストVRAMの構造によってゴミが書き込まれるプレーンが片寄っているだけなのかもしれませんが、何かマウス表示プログラムに問題がありそうです。

さっそく、ROMのマウスカーソルを表示しているルーチンの解析をしてみました。X68030のテキストVRAMは、図4.3.aのように１ビットが１ピクセルに対応する水平ビットマップ方式になっており、複数プレーンを使って色を構成しています。そして、図4.3.bのようにマウスカーソルの灰色の枠にあたるパターンをT２に、青色の中身の部分のパターンをT３に書き込むことでマウスカーソルを表示しています。

この書き込み部分のプログラムは、次のようなプログラムになっていました。

```
1:  and.l d0,(a1)
2:  or.l  d1,(a1)
3:  and.l d0
4:  or.l  d0,(a0)
```

レジスタA１が青プレーンへ書き込むべきアドレスを、レジスタA０が灰色プレーンへ書き込むべきアドレスを保持しており、レジスタD１とレジスタD０がおのおののパターンを保持しています。普通なら、別に問題ないプログラムですが、

青プレーンにのみゴミを出す原因がここにあるはずだ。

と思って見てみる[*1]と、１行目と２行目の青プレーンにandとor命令で連続アクセスしている部分がいかにも怪しそうです。andもorも、いったんメモリをリードしてから論理演算して書き込むという処理になりますから、ここはリード、ライト、リード、ライトときわめて高い頻度でVRAMをアクセスする

＊１
こういうのを、普通は机上デバッグといいますが、私は特に、眼力デバッグと呼んでいます。

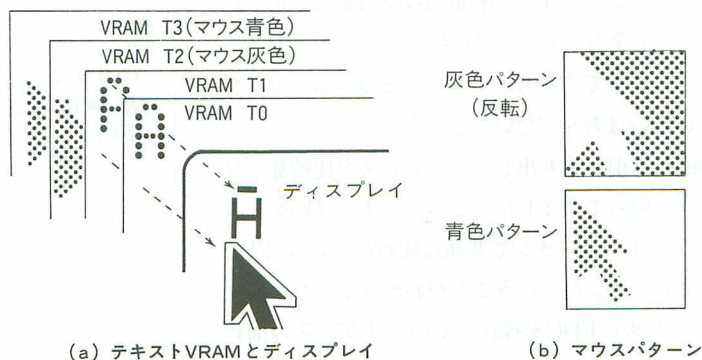



図4.3　マウス表示のしくみ

ことになります。それならばと、ここに当たりをつけてnopを挿入してみる[*1]と、ゴミが書き込まれる現象がピタリと収まりました。

＊1
実はこの対処方法はまぐれ当たりで、後でI/Oアクセスという根本的な問題に対処したときには、nopを抜いてもゴミは出ませんでした。また、これとは別のVRAMゴミ問題が、後で発見されました。とにかくVRAMは問題続きです。

3）see.xでフロッピーをアクセスするとハングアップする

ここではsee.xを取り上げていますが、他のプログラムでもハングアップする可能性はあります。ただ、dirやtypeするだけでは、この現象はほとんど起こりません。

以前はキャッシュオンにすると、dirやtypeを実行しただけでハングアップしていたので透過制御レジスタを使っていましたが、システムへのパッチがしっかりしてきて問題なくアクセスできるようになったので、透過制御レジスタによるスーパーバイザモードでのデータキャッシュの制限を取り払って、これでバッチリだと思っていたのです。

ところが、私がファイルビューワとして愛用しているsee.xでキャッシュオンのときにフロッピーをアクセスすると、確実にハングアップすることが判明したのです。

すでに量産基板の納入が秒読み段階に入っていた時期です。これはさすがにショックでした。

スクロールが遅くなるのは「そういう仕様」として我慢してもらえるでしょうが、キャッシュオンのままうっかりフロッピーをアクセスしてしまうことはありそうです。それで、ハングアップはあまりに強烈です。

「ハングアップの危険性あり、フロッピーはキャッシュオフで使うこと」

という仕様は、さすがに許されません。

ハードウェアの問題かもしれないので、なんとか配布前に原因を突き止めたいところです。他の作業[*2]を中断して全力でデバッグにかかることにします。

＊2
(自称)超豪華取扱説明書の執筆作業をしていました。

デバッグの基本は現象を再現することですから、こういうときはsee.xでつねにハングアップしてくれるというのはありがたいことです。

ひさしぶりに前に作った試験回路を引っ張り出して、アドレスを比較部で見てみると、あるルーチンでループに陥っていました。このルーチンは何かと調べて見ると、IOCSのフロッピーアクセスルーチンで共通に使われている、FDCのステータスチェックをしているルーチンだということがわかりました。

フロッピーをアクセスしにいくためにFDCを操作するのですが、その前にFDCのステータスレジスタを読んで、他の処理を実行中でないかを調べているのです。もし、他の処理を実行中でFDCがビジー状態になっていたら、ア

イドル状態になるのを待つのです。

ここは、次のようなプログラムになっていました。

```
1: loop1:
2:         move.b      (a1),d1
3:         btst.l      #4,d1
4:         bne.s       loop1
5: loop2:
6:         move.b      (a1),d1
7:         bpl.s       loop2
8:         …
```

注：ラベルloop 1、loop 2は便宜上つけたものです。

レジスタA１は$E94001を指しており、これはFDCのステータスレジスタのアドレスです。このステータスレジスタの内容がFDCのアイドル状態に戻らなければならないのに、なんらかの理由で狂ってしまい、いつまでもビジー状態になっていたのです。狂わせている原因がsee.xのどこかにあるはずです。

see.xはX68000の頃のプログラムですから、当然、自前でキャッシュ制御を行っているはずがありません。といって、直接フロッピーをアクセスしているような部分もなく、普通にIOCSコールを使ってアクセスしているようでした。デバッガでトレースをかけながら処理を追っていくと、問題なくアクセスできたりします。

これを調べるのには、本当に骨が折れました。ほとほと困っていたとき、ある現象に気がつきました。デバッガのメモリダンプコマンドでFDCポートをダンプしてみたとき、$E94002のアクセスでバスエラーが起こると$E94001のステータスがビジーになってしまうのです。まるで凍りついたようで、その後、リセットしないかぎり、ステータスが元に戻りません。FDCは８ビットのI/Oデバイスで＄E94001と＄E94003にはレジスタがありますが、$E94002にはないので、ここへのアクセスでバスエラーが起こること自体はおかしくありません。しかし、このアクセスで他の番地が変になるのが不思議です。

しかし、現象からすると、これでビジーになりますから、see.xがなんらかの理由で$E94002をアクセスしてエラーを起こし、FDCのステータスをビジーに凍りつかせているのではないかと考えることができます。

さっそく試験回路の比較部に$E94002というアドレスをセットして試してみると、見事に引っかかってきました。

しかし、see.xを調べてみると、IOCS-$82、B_BPEEK*[1]を使って$E94005番地をアクセスしているところはありましたが、$E94002番地をアクセスしている形跡はありません。ちなみに、$E94005はフロッピーディスクが挿入

されているかどうかを調べるポートのアドレスです。ドライブにフロッピーが挿入されているか、あらかじめ調べているというわけです。

*1
バイト単位のメモリ読み出しコール。

$E94002をアクセスしている奴が、どこかにいるはずだ。

ROMルーチンがわざわざバスエラーになるアドレスをアクセスするような間抜けなことをするはずがないので、原因はsee.xの中にあるはずです。

ここで、はじめてバースト転送の問題に気がつきました。今まで「68040ユーザーズマニュアル」を斜め読みしただけだったので気がつかなかったのですが、68040はデータキャッシュがオンになっていると、バースト転送を使ってアクセスしようとします。

つまり、IOCSルーチンのほうはキャッシュをオフにしてI/Oポートをアクセスするようなプログラムになっていますが、see.x自体はキャッシュオン状態で実行されます。この状態で$E94005をアクセスすると、キャッシュラインの充填のために$E94002を含むバースト転送が行われてしまうのです。

原因はわかったのですが、対策はやっかいです。根本的な解決は、I/Oをアクセスするときにバースト転送が実行されないようにすることですが、そのためにはデータキャッシュがオフになっていなければなりません。see.xにかぎってみれば、IOCSのB_BPEEKでアクセスしていますから、IOCS側でB_BPEEKのルーチンをパッチしてキャッシュオフでアクセスするようにすればすみます。

しかし、I/Oを直接アクセスするアプリケーションでは、みな、同じ問題を引き起こすことになりますから、個々にパッチを当ててI/Oアクセスのときにキャッシュをオフにする命令を挿入しなければなりません。

キャッシュオンの状態でI/Oをアクセスするときは、キャッシュオフにする以外に方法はないのか？

実は、前に透過制御レジスタを使ってスーパーバイザモードのデータキャッシュをオフにする方法を使っていたとき、透過領域の16Mバイトという単位ではなく、もっと小さい単位でキャッシュをオフにするにはMMUを使えばよいということがわかっていました。MMUなら8Kバイト（もしくは4Kバイト）単位でキャッシュを制御できますので、I/O領域だけをキャッシュオフにすることが可能です。

しかし、MMUのプログラムを作った経験はありませんでしたし、68040の

ユーザーズマニュアルを読んでもわけがわかりません。なにやらテーブルを作ってポインタを設定して、といろいろやらなければいけないみたいで、扱いが難しそうで敬遠したくなりました。

いちおう、ためしにMMUを使うをプログラムを作ってみましたが、その時点ではうまくいきませんでした。プログラム自体のミスもあったのですが、実はもっと大きなミスが隠れていました。これについては、後で紹介します。

とにかく、MMUが使えないようではどうしようもありません。前に使って実績のある透過制御レジスタで、とりあえずの対策を施すことにしました。前はスーパーバイザモードではデータキャッシュをすべて禁止にしていましたが、今回はアドレスの上位８ビットが＄FFのとき、つまり、$FF000000～＄FFFFFFFF番地について、データキャッシュを禁止します。この設定をしたうえで、ROMやHuman68k、そしてI/Oを直接アクセスするアプリケーションで使われているアドレスをパッチして、上位８ビットを$FFにしてしまうのです。$00E8xxxxや$00E9xxxxというアドレス部分を、すべて$FFE8xxxxと＄FFE９xxxxに書き換えるのです。個々のアプリケーションプログラムをパッチしなければならない点は同じですが、キャッシュをオフにする命令をいちいち挿入するよりも簡単です。

とりあえず、この方法でI/Oアクセスのバースト転送を防ぐことができました。

中途半端な対応だけど、これで取りあえず回避可能だ。

そして、この対処バージョンを040SYSpatch ver1.3として、第一次配布に添付して配布することになりました。

ハングアップすることを発見したときはショックでしたが、隠れていた問題が見つかったわけですから、よしとしましょう。

しかし、第一次配布後も、続々と問題が発生したのです。

Voice of Users

GCCと040turbo

X68030に040turboを実装していろいろなトラブルをすべて回避した今となっては、もはやGCCのコンパイルは68040ALLコピーバックモードでしか行わなくなってしまいました。

スピードの面では、040turboと80486DX66MHzを搭載したマシンを比べると、どうしても80486DX66MHzを搭載したマシンには勝てないようですが、コンパイラを作り終わるのに1時間とかからないのは、やはり魅力です。GCCの内部では、特に68040だから速くなるような部分はないのですが、「速さは力」やもしれません。

仕事関係のソースも、その対象機（PC-9801やFM TOWNS）では修正せずに、慣れているX68K上でやってしまう私には、ゲームが動かないなどといった問題は障害にはなりません。常時68040モードで動かしています。

キャッシュ関係でおかしくなるプログラム類は自分ですべて修正してしまい、今は完全に68040で動くものしかハードディスクに入っていません。他の仕事が忙しくて、最近の040turboのトラブル関係は自分で再現できていないのですが、別段不便はないので、そのまま使っています。

最近のGCCは、68040にも対応したコードを出せるようにしてありますから、使ってもらえればうれしいですね。最後に、このような高速ボードを企画製作されたBEEPs氏に感謝!!　の一言です。

（文●まりこ　NIFTY-Serve PED00647）

第5章

040turboがやってくる

今度こそ大丈夫?

90度取り付け角度が違った評価基板のせいでだいぶ待たされましたが、いよいよ量産基板ができあがってきます。そろそろ届くかなと思っていたら、フラットパッケージの74AS245が入手できないという連絡が業者から入ってきました。バスサイジングを実現するためのバスの組み換えで1枚の基板に6個の74AS245を使用しますから、第一次配布で数百個を使う勘定です。確かに少ない数ではありませんが、マイナーなタイプではありません。半導体の老舗、テキサスインスツルメンツの石です。

「日本全国手を回したが、手に入らない」というのは担当者の弁。

うーん、2ヵ月前から手配をかけていたはずなのに、おかしいなぁ。

続いて、「74F245なら入る」といってきました。

74F245は、74AS245より若干遅いし、電気も食います。しかし、タイミング的にはそんなにシビアじゃないし、74AS245にこだわってまた待たされるのもかないません。74F245なら、いつ入るのかとたずねてみると、

「もう、買い占めました」との[*1]こと。

いいかなぁと思いつつ、前にGALの品種を15LPから7LPにかえて失敗していますから、大丈夫だろうと思っても、どこに落とし穴があるかわかりません。ここは慎重に、とりあえず1枚だけ74F245を使った基板を作ってもらい、様子を見ることにしました。まだGALが揃っていないということでしたので、それはこちらで用意するということで、とにかく送ってもらうことにしました。

そして、週末には、待ちに待った基板が届きました。

74F245を使っているとはいえ、今度はちゃんとした方向でX68030に取り付けることができました。74F245の動作も問題ありません。だいぶおあずけを食ってしまいましたが、第一次配布分の量産に入ってもらいました。

そして、次の週の金曜日には待望の基板がどっさり届きました。

＊1
間違って74F245を買っちゃっただけじゃないのかなぁとも思いますが、真相は不明。

城之内氏の悲劇

040turbo第一次配布の一番乗りは、同じ愛知県に住む城之内氏でした。彼は、基板が届いた翌日の土曜日に、わざわざ、わが家まで基板を取りにきてくれました。

前日届いた基板を取り出し、X68030に取り付けてテストします。わが家のX68030は両タワーともカバー開放*1ですから、基板の取り付け・取り外しは自由自在です。

＊1
今でも、カバーは全開状態です。

うまく立ち上がらなかったらどうしよう。

そんな不安が一瞬頭をよぎりますが、パワーオン。68040を示すLEDが赤く光り、いつものようにすんなり起動しました。

まだテスト方法を考えていなかったので、とりあえず、適当にコマンドを叩いてみます。SX-WINDOWを起動させ、負荷がかかりそうな動画再生を何本か走らせながらEasy-Drawでベジェ曲線をウネウネ変形させてみたりします。

サクサク動いて、城之内氏も満足そうです。

よかった。

前日、やっと仕上げた040turboの取扱説明書のコピーと、パッチプログラム類のフロッピーをコピーして、

「はい、第1号。NIFTYに報告書き込んでね」

といって手渡しました。

前日、取扱説明書の最後の仕上げで朝帰りだったので、再び布団に潜り込みました。

夕方、さて、どうなったかなとNIFTY-Serveをのぞくと、恐怖のメールが入っていました。

68040モードにしてもLEDが点灯しない！

とのこと。

テストしたときは確かに点灯していたはずだ。おかしい。

クロックの配線がミスっているとLEDが点灯しませんから、取り付けミスではないかと思い、とりあえず返事を書いておきました。

しかし、これは別の原因によるものでした。他の基板を調べてみると、なんと今回届いたLEDケーブルの配線がすべて逆だったです。LEDは極性がありますから、逆方向につないでいると発光しません。前の週に届いた74F245を使った量産版のテスト基板に添付されてきたケーブルはちゃんとつながっていたのに、今回届いた分はことごとく逆になっていました。

城之内氏に渡す基板をテストするとき、ケーブルを取り出すのが面倒だったので、前の週のテストで使ったケーブルを使っていたのです。テストミスといっていいでしょう。しかし、まさかケーブルの配線を間違える*1とは思っていませんでした。城之内氏には気の毒なことをしましたが、大量配布する前に見つかったのはラッキーだったといえるでしょう。

残りのケーブルを１本１本すべて直し、以後、基板のテストは１つずつケーブルをつないでテストすることを心に誓いました。

ケーブルを直して、やっと040turboが使えるようになった城之内氏からの記念すべき第一報を紹介しておきましょう。

*1
その後、第二次配布分でも、やはりケーブルが間違って入ってきました。

682/999　GGB00312城之内　　040TURBOボード使用レポート（その０）
(14)　　93/10/25 00:09

040TURBO快速

なんとか動くようになったので報告まで。
土曜にBEEPs氏のお宅で受け取り、試行錯誤しながらやっと起動できるレベルになりました。JUNKにアップされたFLOAT040.xも使わせて頂いております。
これから040TURBOのポテンシャルを引き出せたら、と思います。
まだ、起動して動いている程度ですから、これから調整が必要です。
(HENWIN ではプロテクトがかかっているとか言われるし)
では、また。

p.s.SXウィンドウで書いているのですが、当たり前のようですが030より軽い
　030に戻れなくなったりして。　(笑)

Written by 城之内

第一次配布のプログラム

とりあえずシステムへのパッチを優先して、040SYSpatch.sysを作ってなんとか使えるようになってきました。では、アプリケーションのほうで問題がないかとHuman68kのシステムディスクに入っている実行ファイル*1を調べてみたところ、format.x、fastio.x、fsx.xのなかにキャッシュ制御を自前で行っている部分が見つかりました。

このなかでformat.xに入っている部分は、SCSIハードディスクに書き込まれるSCSIのデバイスドライバ「SCHDISK」*2本体のようなので除外すると、残るはfastio.xとfsx.xです。

どちらも真面目に解析するには骨が折れそうなプログラムなので、安直に68030のキャッシュと思ってキャッシュ制御している部分を、68040のキャッシュ制御になるように変更するだけにしておきました。不都合が出たら、そのときに考える*3ということで、よしとします。

これら2つのプログラムのバイナリ差分を用意して、第一次配布に備えていました。

patexec.sysができてくる

patexec.sysは、実行ファイルそのものにパッチするのではなく、そのファイルがメモリ上にローディングされて実行されるときにメモリ上でパッチをあててくれるというプログラムです。もともとはX68000とX68030で実行ファイルを共有するため*4のプログラムでしたが、これは040turboにちょうどいいということで、作者の中村ちゃぷに氏に68000と68030の対応だけでなく、68040にも対応してくれるよう、頼んでいたのでした。

そして、68040対応バージョンが、第一次配布の直前にバイナリメールで届いたのです。これを使えば、68040対応にパッチしたfastio.xやfsx.xの別ファイルを用意しなくても、68030用の実行ファイルを68040でも使えるようになります。

さっそくバイナリ差分はやめて、patexec.sys用のパッチファイルを用意することにしました。あわせて取扱説明書の説明文も修正*5しました。

2つの浮動小数点演算エミュレータ

68030と68040の互換性という点では、キャッシュの問題以外はたいしたこ

＊1
フリーソフトウェアについては特にチェックしませんでしたが、自前でキャッシュ制御するようなアプリケーションはそうあるとは思えなかったので、後回しです。

＊2
040SYSpatch.sysのほうで起動時にパッチを当てます。

＊3
fastio.xのほうは不都合が起きるとまずそうですが、私は使っていないのであまり深刻に考えていません。

＊4
普通こんなことはあまりしないと思いますが、作者の中村ちゃぷに氏は、2台のマシンをSCSIの両端につないでハードディスクを共有していたために、これを作ったそうです。

＊5
これが配布の前日の徹夜の原因の1つです。

とはありませんでした。しかし、68040が68882の浮動小数点演算のサブセットということから、68030+68882の組み合わせで実行できるものが、68040では実行できないという問題が起こります。

一番大きな問題は、X68シリーズの浮動小数点演算パッケージfloatのX68030版float4.xが使えないということでした。ソフトウエアエミュレーションするfloat2.xを使う*1ことはできましたが、せっかく浮動小数点演算命令があるのに68882のフルセットを揃えていない理由で遅いfloat2.xを使わなければならないというのは残念です。

さらに、数は少ないですが、68030+68882の組み合わせをターゲットにして、直接、浮動小数点演算命令を使ってプログラミングされたアプリケーション*2は、68040でエラーになってしまいます。もっとも、68030+68882をターゲットにしたプログラムはX68030専用になってしまいますから、X68000がいまだ主流の現状では数えるほどしかありませんので、我慢できなくはないのですが、浮動小数点演算性能を追求して専用プログラムの道を選んだこれらのアプリケーションが040turboでより速くなるのを見たいところです。

私は、浮動小数点演算のプログラミングはやったことがなかったので、第一次配布の参加者に期待していたわけですが、期待どおり対応プログラムが作られました。それも、2種類、別々のアプローチがとられたのです。

1つは、floatを改造して68040対応にした鈴木国文氏のfloat040.xでした。float4.xよりも当然高速です。

もう1つは68040でサポートされなくなった浮動小数点演算命令をエミュレーションする、中村ちゃぷに氏のpfloat.xです。

float040.xとpfloat.xの2つの浮動小数点演算プログラムですが、性質はだ

＊1
040turboの開発をしているときは速度は二の次だったので、これで十分でした。

＊2
私の家にはDōGA－CGAシステムのREND30.XとHat氏のレイトレーシングソフトくらいしかありません。

C O L U M N

取扱説明書

040turboの配布で一番凝っているのは、実は取扱説明書といってもいいでしょう。

最初は、簡単な取り付け方法の説明書を数枚つけるだけのつもりだったのが、書いてるうちに、どんどんノってきてネジの種類まで調べだし、図解入りのX68030の分解方法と040turboの取り付け方を延々10ページにわたって説明しています。

こうなってくると、040turboのハードウェアの説明も欲しいな、ソフトウェアも必要だなと、最後には68040のプロセッサと68030の相違点にまで言及し、60ページを超える大作となってしまいました。

もったいないので、本書の付録にも収録しています。

いぶ違います。float4.x+pfloat.xの組み合わせでfloat4.xを使うこともできるようになりますが、float040.xのほうがオーバーヘッドが少ないので、浮動小数点演算パッケージとしてはfloat040.xのほうが得です。

一方、pfloat.xを使うと、68030+68882をターゲットにした専用プログラムが使える*1ようになります。

両方登録しておけば、鬼に金棒ということです。

こうして、頭の痛い問題だった、浮動小数点演算関連の問題も解決しました。

GCCも届く

GCC*2は、メーカー純正のXCよりもあらゆる点で優れているため、X68ユーザーのほとんどがこちらを使っているのではないかと思いますが、このX68版GCCの事実上の標準と思われる「まりこ版GCC」の移植者、まりこ氏が、040turboの第一次配布に参加表明してくれたので、さっそく、68040への対応をお願いしていました。

68040対応といっても、GCCのコンパイラ自体は、68040でもそのまま動きますし、普通にコンパイルしている分には、生成したオブジェクトも68040で動きます。

問題となるのは、68030+68882をターゲットとしてコンパイルしたオブジェクトです。この場合は、浮動小数点演算命令の問題*3で、68040では使用できません。かといって、68882を使わない指定にしてコンパイルすると、68040で使用できる浮動小数点演算命令もいっさい活用できなくなるので、効率が悪くなります。

68040をターゲットとして、68040で使える命令だけを使ったオブジェクトを生成する機能をGCCに追加してもらうように、お願いしたわけです。

こちらも、第一次配布の直前にテスト版*4が届きました。

ためしに、X68030で68882を搭載したときにそのスピードに感動したレイトレーシングプログラムを、このGCCで68040対応としてコンパイルして実行してみたところ、キャッシュオンで35秒。68030+68881でも驚いていましたが、さらにその約2倍の速度になりました。

＊1
まだ若干問題はありますが。

＊2
いうまでもないことですが、「GNU C Compiler」のことです。

＊3
もちろん pfloat.xを使えば実行できますが、ソースファイルがあるプログラムを自分でコンパイルするなら、最初から68040対応のオブジェクトをgccに作ってもらったほうが得です。

＊4
今では、正式な公開バージョンで、68040対応が明記されています。

さっそくバグ

順当に配布がすすみ、何人かの手に040turboが渡っていくと、さっそく、040SYSpatch.sysにバグが見つかりました。

ver1.3だとキー入力をまったく受け付けないというものでした。保険のためにと思い入れておいたver1.2*1のほうは幸い使えたので、いきなり使えないという情けない事態にはならずにすみました。

＊1
「ハングアップの危険性あり、フロッピーはキャッシュオフで使うこと」というバージョンです。

この現象は私のX68030では起こらないため、調査に手間取っているうちに参加者の１人、PUNA氏が原因を突き止めてくれました。ver1.3では、I/Oアクセスのアドレスの上位８ビットを$FFにするために、$00E9xxxxというデータをすべて$FFE9xxxxに書き換えていたのですが、関係ないところまで書き換えていたようです。さっそく修正版ver1.3aを作り、NIFTY-Serveにアップロードしました。この後、PUNA氏は、Human68k ver3.02への対応もしてくれました。

ちなみに、Human68k ver3.xxには、現在、次のバージョンがあります。

X68030に添付されているもの	ver3.00
SXデスクアクセサリ集に添付されているもの	ver3.01
SX-WINDOW ver3 に添付されているもの	ver3.01
XC ver2.1 NEW KITに添付されているもの	ver3.02

オリジナルの040SYSpatch.sysはver3.01にしか対応していませんでした。Human68k ver3.02はじかにキャッシュ制御命令を使ってプログラミングされていた部分が１ヵ所を除いてIOCS-$ACのキャッシュ制御コールを使うように修正されていました。こうなっていれば、IOCS-$AC側のルーチンを68040に対応させるだけで、これを呼び出している側にはパッチが不要になります。

残る１ヵ所はアプリケーションプログラムの命令書き換えを行っているために、書き換え後に命令キャッシュをクリアするために使われていました。

この命令書き換えはちょっとおもしろいので、紹介しておきましょう。

特権命令の変更

この命令書き換えは、68000ではユーザーモードで実行できたステータス読み出し

```
move from SR
```

が、68030ではスーパーバイザ命令になったために、ユーザーモードでは特権命令違反になってしまうことへの対処です。SR（Status Register）は、演算の実行結果によるアンダーフローやオーバーフローなどのフラグを保持するCCR（Condition Code Register）部分と、割り込みマスクなどのハードウェアの動作に密着した部分があります。68000では、SRへの書き込みはスーパーバイザモードでなければできません[*1]が、SRからの読み出しはユーザモードでもできた[*2]のです。

ところが、68010以降は、読み出しもスーパーバイザモードでしか実行できなくなりました。ユーザーモードで実行すると、特権命令違反になってしまいます。このままだと演算結果のフラグを保持するCCR部分が読めなくなりますので、この部分のみを読み出す

```
move from CCR
```

という命令が新設[*3]されました。

しかし、68000時代には使えた命令が68030で使えないのでは、X68000用のプログラムが実行できなくなります。

これへの対処としてHuman68k ver 3では、特権命令違反のサービスルー

＊1
CCRへの書き込みは、ユーザーモードでも可能なmove to CCRという命令があります。

＊2
コンディションコードレジスタ部分を読むためのmove from CCRという命令自体が存在しなかったので、ユーザーモードでmove from SRが使えるようになっていたのでしょう。

＊3
そもそも、この命令が68000になかったのが間違いなのですが。

COLUMN

68シリーズの互換性

モトローラは、プログラムの互換性に対する意識が希薄な気がします。スーパーバイザモードはOSのためのもので、ベンダーが提供するから、ユーザーは気にしなくていいという方針だからなのでしょう。

この点、インテルの互換性への並々ならぬこだわりとは対照的です。

インテルのプロセッサは、8086時代のソフトウエアを実行できるモードをつねに設けるアッパーコンパチブルな設計になっています。これがまた、「美しくない」という悪い評価の原因でもあるのですが、世の中の情勢を見た場合、ソフトウエア資産は美しさに勝るといえそうです。

チンのなかで命令コードを調べ、「move from SR」を実行しようとして特権命令違反になっていた場合は、アプリケーションプログラム自体の機械語コードを「move from CCR」に書き換えて再実行しているのです。

ここで命令キャッシュが生きていると、たとえメモリ上の機械語コードを書き換えても、キャッシュ上は前の命令のまま残ってしまうので、いったん命令キャッシュをクリアする必要があります。プログラムの書き換えをせず、エミュレーション*1するようになっていればキャッシュ制御をしなくてもすみますが、毎回、この処理が入ることになりますのでスピード的に不利です。そこで、プログラムを修正することにしているのでしょう。プログラム自体を書き換えるというのはあまりきれいなやり方ではないのですが、2回目以降は特権命令違反にはならないので、うまいやり方*2です。

話が脇道にそれましたが、Human68k ver3.02では、この部分にまだキャッシュ制御命令が使われているのです。特権命令違反の処理のなかでIOCSコールをするのを嫌ったのかもしれませんが、ここにもIOCS-$ACを使ってくれれば、Human68kはノーパッチですむようになります。次のバージョンアップではどうなるか、興味のあるところです。

それにしても、040SYSpatch.sysのバージョンアップが、さっそく040turboの参加者の手で*3行われたのは心強いことです。

＊1
サービスルーチン内でSRの内容を読み出し、読み出し先のレジスタに格納してやればよいのです。

＊2
Macintoshも互換性をとるために、このようなことをやっているそうです。

＊3
もともと、それを期待して基板配布に踏み切ったのですが。

COLUMN

X68シリーズの互換性

68000と68030の互換性の問題は、「move from SR」にとどまりません。一番インパクトがあるのが、各種の割り込み発生時にスタックに積まれるデータフォーマットの変更でした。

X68では、$Fxxxという未定義命令をDOSコールの目的で使用しており、引数をスタック経由で渡すようになっていましたが、68030でスタックに積まれるデータフォーマットが異なったため、引数を取り出すオフセットアドレスが変わってしまいます。

このため、Human68kでは、プロセッサが68000か68030かで動作を変えて、この違いを吸収しているわけです。

また、そもそも$Fxxxxという命令が68030ではコプロセッサ命令になっているため、場合によっては未定義命令にならずに特権命令違反になってしまうこともありますが、この違いもHuman68kで吸収するようになっています。

このように、Human68kの努力によって、68000と68030の違いを吸収しているのです。

第一次配布の反応

さて、040turbo第一次配布者の反応ですが、040SYSpatch.sys ver1.3にバグがあったとはいえ、みんな好意的でした。もっとも、個人で作ったものですから、参加するほうも駄目でもともとだったのかもしれません。

```
698/999  GGA00464そると                    お家に040
(14)     93/10/26 22:41

 今日帰宅したら届いていました。    (^_^)

 早速インストールが終了したところですが、どうやら動いている
ようです。    (^_^;;  現在、多少の不具合いがありますが調整して
いきたいなと思っています。

 長時間動かしているわけではないので何とも言えませんが、クロッ
クが34.8MHzでも固有の問題があるものの何とか動いているよう
です。ちなみに、pv.xの出力では初代比22倍程度と出ます。    (^_^)

 クロック切り替え回路を付けていると、クリップでのクロックの
引き出しが困難なので直接半田付けする必要があります。

 BEEPsさんご苦労様です。

 そると
```

```
701/999  GBH00172鈴木 国文          RE:040TURBO発送しました
(14)     93/10/27 00:22  696へのコメント

BEEPsさん、徹夜の作業お疲れ様です。
040SYSpatchが旧バージョンだった以外は特に問題なく動作しています。

やっぱ、速いですねぇ〜
最近レイトレに凝ってるので非常に楽しみであります。

自画自賛になりますがFLOAT040.Xも順調・・・80%くらいは順調で
無改造X68030+882と比較して・・・
```

各種演算を100000回ループ

	FLOAT2.X	FLOAT4.X	FLOAT040.X
単精度加算	3350msec	2900msec	600msec
単精度乗算	3350msec	2520msec	690msec
倍精度加算	3640msec	3390msec	1280msec
倍精度乗算	6180msec	3700msec	1290msec
単精度平方根	13400msec	7030msec	2240msec
倍精度平方根	9740msec	2790msec	1710msec
倍精度cos x	11970msec	5920msec	5100msec
倍精度log x	11590msec	4970msec	4080msec

単精度加算が4.8倍、倍精度で2.6倍をはじきだしてます。
三角関数と対数関数があまり上がってないのは私の怠慢です。_(_)_

うぅ～ん、もっともっと速くしてやるぅ～　(T^T)

鈴木　国文

このほかにもたくさんの書き込みが NIFTY-Serve FSHARP1「ハードウェアの部屋」を賑しました。

一方、ある程度予想していましたが、やはりクロックアップしたマシンでは調子が悪いようでした。

722/999　JBG03507大浦義宏　　040 TURBO 34.8MHz 起動しました
(14)　　93/10/30 19:51

みなさん　040 TURBO34.8MHz動作について色々とありがとうございます。

何かと初心者でいたらない事もありますが、とりあえず皆様にレスを・・・

そるとさん

＞－ CPU CLOCKを34.8MHzにすると68030モードでの動作が不安定。

僕は初めICクリップで接続したのですが、起動しませんでした。(^^;)
#そこで半田で両方とも直にくっつけたところうまくいきました。

#ですが、JPEGL.Xなんかを使うと画面が15KHzになってしまったので
#配線をめいいっぱい短くして接続したらうまくいきました。

＞－CPU CLOCKを34.8MHzにすると68040で画面にアクセスする
＞と画面が乱れまくる。

#SXWINDOW でCGを表示すると画面が乱れる症状が出ました。

BEEPsさん
#ver1.3aを使用したところ無事キー入力でき、パッチにも成功しました。
(^^) v
最後に、X68030の030モードで今まで通りPOWER.Xの動作で385%を表示しております。また040命令キャッシュONで365%の表示を出しました。
どうやら、なんとか起動したみたいです。嬉しいのはやはりSXWINDOWがさくさく動く事です。ついでにJPGファイルもサクサク表示です。
うーん、もったいない使い方で恐縮です。(^^;)

これでオーガスタなんか動くと・・・バキッ！！　　大浦義宏

草の根ネットで唯一、私が出入りしているMAX-BBS*[1]でも、040turboの参加者を募っており、こちらでも活発に報告がされました。

*1
前に川崎にいたとき、X68ボードのSigopをしていた付き合いで、名古屋にきた今もアクセスしているのです。

=-=
(#X68　/#18) [5833/ 5838] 93/10/31 13:54:44　　588byte
MAX0034 [Ya.O]　　040Turboのこと
=-=
　ふみぃ、本当に熱を持ちますね。　某マップで2300円くらいで040用の

ファン付きヒートシンクが売っていましたので作るの面倒な私は買って来

ましたけど。　ちゃんとつめがあって便利

でも、ファンつけないと結局ヒートシンク自身があっちち。

　うちは、クロックアップしてますから、マウス表示でゴミがでまくり特

に右ボタンで線が走ります。マウス表示のウェイト量を増やしてみました

が効果ないようだし、25MHZではBEEPsさんのパッチで問題ないし。。。

テキストメモリが追い付かないのかしらん？　でも、文字表示だけなら問

題ないしなあ。 25MZHに戻そうかしらん？

や〜お

=-=
(＃X68 /＃18) [5834/ 5838] 93/10/31 16:15:55 1213byte
MAX0051 [Y.] Re:040Turbo のこと
=-=

＃＃＃ #X68 p5833でMAX0034 [Ya.O] さんがいいました。
> 2300円くらいで040用のファン付きヒートシンクが売っていましたので
> 作るの面倒な私は買って来ましたけど。 ちゃんとつめがあって便利

私も、買いました。某マップでではないけれど。ただ、爪のついている向きがあまり良くなくて．．．GALもかなり熱くなるんで、スリットが縦になるように取り付けると（風がGALに当たるように）下に抜け落ちる方向になるんですよねぇ。仕方ないんだけど。取敢えず大丈夫だし。(右下のGALには、例によって386を張り付けてある。(笑))

> うちは、クロックアップしてますから、マウス表示でゴミがでまくり
> 特に右ボタンで線が走ります。マウス表示のウェイト量を増やしてみま
> したが効果ないようだし、25MHZではBEEPsさんのパッチで問題ない
> し。。。

うちでは、ゴミの問題は落ち着いたよ。Ya.Oさんはクロックいくつ？
私は、SYSTEM CLOCK 25/MPU 33MHzでコンソール、SXともにゴミは出なくなりました。040modeの時はスタティックカラムモードにする様にしてます。ほとんど、タイミングの問題みたいですね。
030modeでスタティックカラムモードにすると、33MHzではバスエラーになるんで、この時はSW1をショートするように、二連のSWつけてます。
パッチのバージョンは1.3a使ってます。（あっ、あと例のHIOCSね (笑)）

Y.

私のマシンは、下手に改造して条件が変わると嫌なので、まだクロックアップしていません。このため、クロックアップした場合に出る現象がつかめないため、不具合にどう対応したらよいかわからないので、とりあえずクロックアップしている人[*1]は自分でどこまでなら耐えられるか見極めて、飼い慣らしてもらうしかありません。

＊1
X68000ユーザーにとって、クロックアップは「お約束」のようなものですから、今はある程度対応しました。

COLUMN

クロックアップ

私の記憶が正しければ、FIRST.R氏がX68000 XVIを16MHzから24MHzにクロックアップしたNIFTY-Serveでの報告が発端だったと思います。

これは、画期的な報告でした。X68000 XVIのクロック16MHzというのは、もともと68000のクロックが16MHzまでしかないためにこの値となっているので、この点でメーカーを責めるわけにはいきませんが、たかだか1.6倍の性能アップは、はっきりいって、多くのX68000ユーザーを失望させるものでした。

ところが、ハードウェア改造とはいえ、オシレータを取り替えるという比較的簡単な作業で、10MHzのX68000の2.4倍になるわけです。

中途半端に思われたX68000 XVIが一躍脚光を浴び、X68000からの買い替えが加速されたといってもいいでしょう。さらに「今後のロットは、クロックアップができないよう対策が施される」といったデマが流れ、これを煽りました。

さらにおもしろいのは、X68000 XVI以前の10MHz機種でもクロックアップへの試みがなされました。そもそも、プロセッサと周辺のクロックが分離されていない従来機種では、プロセッサのクロックだけを上げることができないと思われていました。ところが、プロセッサといっしょに周辺のクロックが上がってもソフトによっては動くということがわかり、従来機種のクロックアップも流行しだしたのです。

この貪欲なまでのクロックアップ熱は、25MHzのX68030にも容赦なく及び、X68030が発売開始されると、すぐにクロックアップの報告がなされました。実際、33MHzあたりにクロックアップするのであれば余裕で動いているようです。

ちなみに、私が今まで耳にした限界は50MHzでもIPL画面だけは出たというものでした。

ほーっ、これが040turboか、
思ってたよりきれいな基板じゃん。
040 turbo

MMUと大チョンボ　040SYSpatch.sys ver2.0

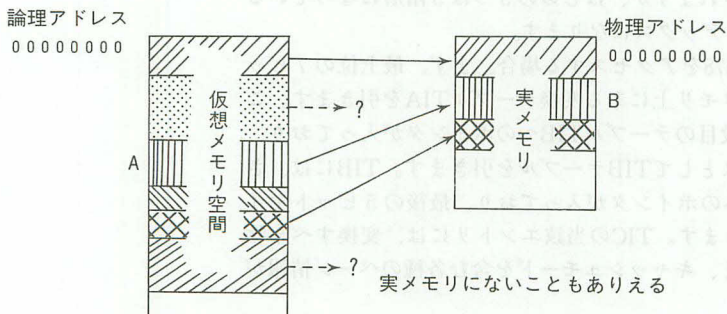
配布作業も一段落ついたので、いよいよ、積み残しだったMMU（Memory Management Unit）によるI/Oエリアのキャッシュオフの制御について検討を始めることにしました。

本来MMUは、図5.1のように論理アドレスと物理アドレスを変換するためのもので、A番地をアクセスするようにプログラムを作っていても、MMUによって実際にはB番地がアクセスされるようになるのです。これは、仮想記憶[*1]を実現するためにはなくてはならないメカニズムです。

68040のMMUは、「ページ」と呼ばれる、８Kバイトもしくは4Kバイトのアドレス空間を１つの単位として、アドレス変換を行うことができるのですが、付加機能としてページごとにキャッシュのモードを指定することができます。本来のアドレス変換[*2]の目的ではなく、この付加機能を使いたいためにMMUを動かすというわけです。

さて、MMUの実際の使い方ですが、これが結構面倒です。コラムに簡単にMMUの動作を紹介しましたが、興味があったら、68040のユーザーズマニュアルも見てください。

I/Oエリアにあたるページにキャッシュ禁止の設定をした変換テーブルを用意し、MMUを有効にすれば、I/Oへのバーストアクセスが回避されるというわけです。

しかし、真面目にMMUを使うプログラムに再挑戦したにもかかわらず、

＊1
メモリ内容をハードディスクなどの二次記憶装置に一時的に退避したり戻したりすることで、実際のメモリ量よりも大きなメモリがあるように見せる方法です。

＊2
もともと68000をターゲットにして作られたHuman68kですから、ver 3になっても仮想記憶をサポートしていませんし、X68030のプロセッサが68EC030というMMUなしのプロセッサですから、今後もMMUを使えるようになる可能性は少ないでしょう。

図5.1　MMUの働き

いくらやってもうまくいきません。MMUをイネーブルにしたとたん、暴走してしまうのです。

040SYSpatch.sys ver1.3の頃に試したときは、MMUへの理解が浅くて自信がなかったのですが、今度はちゃんと勉強しましたので、MMUの制御プログラム自体は間違いないはずです。ソフトウェアに間違いないと確信できるようにならないとハードウェアを真剣に調べないというのも困ったもの[*1]ですが、たいてい、そんなものです。

そして、恐れていたとおり、ハードウェアのミスを発見してしまいました。68030は、FC 2～FC 0信号（Function Code）によって、メモリアクセスがスーパーバイザ空間なのかユーザー空間なのかといった情報を示します。これに対し、68040は、FC 2～FC 0信号線そのものはなくなってしまいましたが、似たような意味を持つTM 2～TM 0信号がありますので、変換回路では、この信号からFC 2～FC 0相当の信号を作り出しているのです。

しかし、このTM 2～TM 0信号には、68030にはないテーブルウォークというメモリアクセスの状態も出力されてきます。名前のとおり、MMUがテーブルウォークしているときのメモリアクセスを示しているのですが、この部分のGALのソースリストは次のようになっていました。

＊1
実際、ハードウェアのせいだといわれてさんざん調べたあげく、実はソフトウェアのバグだったということは仕事で嫌というほど経験しています。

COLUMN

MMUのアドレス変換

ページサイズが8Kバイトの場合、68040のアドレス32ビットは7ビット＋7ビット＋5ビット＋13ビットに分けられます。最後の13ビットはそのまま物理アドレスとして出力されますが、はじめの3つは3階層になっているMMUの変換テーブルのインデックスになります。

具体的にいうと、$12345678をアクセスする場合、まず、最上位の7ビットをインデックスとして、メモリ上にある変換テーブルTIAを引きます。このTIAの各エントリには2段目のテーブルTIBへのポインタが入っており、次の7ビットをインデックスとしてTIBテーブルを引きます。TIBには、さらに3段目のテーブルTICへのポインタが入っており、最後の5ビットをインデックスとしてTICを引きます。TICの当該エントリには、変換すべき物理アドレスの上位19ビットと、キャッシュモードを含む各種のページ情報が入っています。

MMUのSRP（Supervisor Route Pointer）およびURP（User Route Pointer）には、この1段目のテーブルとなるTIAのアドレスをセットしておきます。後は、MMUが「テーブルウォーク」と呼ばれるモードで必要に応じて変換テーブルを勝手に引いてくれます。

```
 1:   TABLE tm040 => fc030{
 2:           'b'000 => 'b'111;
 3:           'b'001 => 'b'001;
 4:           'b'010 => 'b'010;
 5:           'b'011 => 'b'111;     <===   テーブルウォーク
 6:           'b'100 => 'b'111;     <===   テーブルウォーク
 7:           'b'101 => 'b'101;
 8:           'b'110 => 'b'110;
 9:           'b'111 => 'b'111;     <===   CPU空間
10:    }
```

左側のtm040というのが68040のTM2～TM0のコードで、これがきたとき、右側のfc030のようなファンクションコードをFC2～FC0に出力するようになっているのですが、5行目と6行目のテーブルウォーク（011, 100）がCPU空間（111）に割り付けられていたのです。

このCPU空間というのは、割り込みアクノリッジやコプロセッサのアクセス[*1]などに使われる特殊なメモリ空間です。テーブルウォークをCPU空間に割り付けていたため、MMUを有効にしても通常のメモリ上に用意されたMMUテーブルを読めないでバスエラーになっていたのです。バスエラーになってもエラー処理ルーチン自体が動けませんから完全なハングアップ状態です。

わかってしまえば簡単なことでした。この変換を行っているIC4のGALを直して試したところ、MMUをイネーブルにしてもハングアップしなくなり

＊1
前に出てきた68040から68882を使うという方法は、この空間を使っています。

COLUMN

メモリアクセスのたびにそれぞれのテーブルを引くのはオーバーヘッドになるので、MMU変換専用のキャッシュも、普通の命令キャッシュ、データキャッシュとは別に内蔵されています。

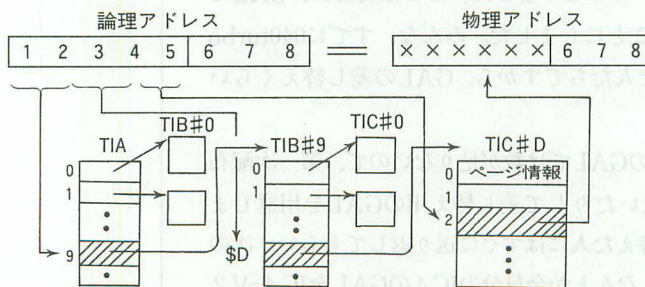



図　a．アドレス階層　b．ページテーブル

ました。そのうえ、MMUをイネーブルにしてすぐにわかったのですが、スクロールが驚くほど軽快になったのです。キャッシュオンでスクロールが遅くなるのはI/O系のアクセスタイミングの問題と思っていましたが、こっちもバーストアクセスが原因だったわけです。see.xでフロッピーをアクセスしてもハングアップしないのはいうまでもありません。

画面出力が速くなると使用感はずいぶん変わってきます。いかに68040のMMUが優秀といっても、テーブルウォークのオーバーヘッドは入っているはずです。しかし、そんなことをまったく感じさせません。むしろ一皮むけたという感じです。

ついでに、FDC割り込みルーチンでのキャッシュ制御やマウス表示のゴミ対策など、個別に対処していた問題も、このMMUによるキャッシュ制御を取り入れることですべて不要になっていたのですが、この時点では、まだそこまでは気が回りませんでした。

ver2.0とGALの差し替え

MMUを使用してI/O領域をキャッシュ禁止にするようになったので、ver 1.3でやっていたI/Oアクセスのアドレスの上位8ビットを$FFにするパッチもすべて削除し、040SYSpatch.sysを新たにver2.0としてNIFTY-Serveにアップロードしました。

しかし、頭が痛いのはハードウェアのほうのバージョンアップです。こちらは簡単にはすみません。せめてもの救いは、このハードウェアのバグが基板自体の改造には及ばず、GALの差し替えだけですんだ[*1]ということです。普通、ハードウェアのバージョンアップといったら、基板を送ってもらって差し替えて送り返すのでしょうが、送料もバカになりません。ここは安易に、GALだけ送って各自で差し替えてもらうことにしました。みんな、すでに040turboの実装のために68030を差し替えた人たちですから、GALの差し替えくらい朝飯前でしょう。

GALを送るといっても、手持ちのGALでは数が足りないので、第一次配布で残っている基板からもGALを抜いたりして差し替え用のGALを用意します。それでも足りないので、差し替えた人にはすぐに送り返してもらい、次の人に送るということを繰り返して、なんとか全員分のIC 4のGALをIC 4-V 2にバージョンアップすることができました。

＊1
結局、後で基板改造に及ぶ問題が出ましたが。

本領発揮?

自分でいうのもなんですが、MMUを使うようになった040SYSpatch.sys ver2.0は予想以上の効果で、I/Oアクセスの問題がきれいに解消され、68040の動作はすこぶる快調になりました。みんなのもとにIC4-V2が届くにつれて、NIFTY-Serveにも続々と書き込みが行われるようになりました。

```
776/999　GGA00464そると　　　　　　　　040TURBO
(14)　　93/11/04 23:04　753へのコメント

　今日新しいGALが来たので早速取り付けました。　cache on id
の状態だとムチャクチャ速いですね。　　(^_^)

　これでコプロ命令をエミュレートするプログラムで　rend030.x
が動けば...　(他力本願　(^_^;;)

　そると
```

```
777/999　GBF00222PUNA　　　　　040TURBO新GAL
(14)　　93/11/04 23:04

本日新GALが届きました。そこで 040SYSpatch ver2.0 を使ってみまし
たが、なかなか快調ですね。SX-WINDOWで画面にゴミが出ていたの
がぴたりとおさまりました。

ということで 040SYSpatch ver2.0 の Human ver3.02 対応版をJUNK
にアップしておきました。

PUNA
```

```
778/999　KHF03720　なっち　　　　　　　RE:040TURBOその後
(14)　　93/11/04 23:12　774 へのコメント　コメント数：　1

新しいGALを取り付けました。(古いGALを取り外す時に2～3本足を曲
げてしまいました。一応元に戻しておきましたが・・・。いつも面倒なこ
```

とを起こしてしまって申し訳ありません。)

今まで命令キャッシュ、データキャッシュともONにした状態でSXWIN
を立ち上げると、マウスの右ボタンを押した時に画面が乱れていたのです
が、その現象がなくなりました。今のところ両方のキャッシュON状態で
作業していますが、問題は発生していません。(^_^)(ちなみにadjust2.r
を使用して816×620位のドット数でSXWINを使用しています。)
旧GALではキャッシュOFF状態でも鳩時計が「おかしな命令」かなにか
で実行出来なかったのですが、新GALでは動作するようになりました。

なっち

冷静に考えれば、ver1.3がふがいないだけだったのですが、こういったメッセージが書き込まれると、差し替え用GALの地道な発送作業の苦労も報われます。

68030のピンチ

クロックアップは別として、おおむね良好かなと思っていたら、思わぬ落とし穴が待っていました。

744/999　NBH02724　じゃぎゅあ　　　040TURBOが来ました
(14)　　93/11/02 21:23　　　　　　　コメント数：　1

ちょうど、486用冷却ファンと68030-25を買った日でした。

68040モード(キャッシュオン)で ed.r 0.99XC が起動しないのが
ちょいと不便なくらいで、速い。
コンパイルが速い。2.5倍にはなっています。
といったところで、040モードで「ぴーんち」に陥ったことは
今のところはありません。

それとは反対に、030モードが「ぴーんち」です。

「おかしな命令」「アドレスエラー」が頻繁に出てしまいます。アドレスエ
ラーは「おかしな…」が原因だと思われます。

2500円の68030-25が原因かと思ってX68030付属のEC030に交換しても同じで、040TURBOを外して68030-25にすると正常でした。
常駐したものがFLOAT2.Xのみでも同じようにコケてしまいます。

じゃぎゅあ

配布した040turboは、すべて私のX68030で68040モードも68030モードもテストしてから発送していますから、初期不良は考えにくいものです。それに、68040側は変換回路が入っているため問題が起こる可能性がありますが、68030側はCPU切り替えのための数本の信号線しか操作しておらず、ほとんどの信号線はそのままX68030の本体側とつながっています。このため、68040モードが正常に動いているなら、信号線の断線といった問題は考えられません。

040turboを搭載することで信号線の長さや負荷が増えますから、何もない68030よりは条件は悪くなるでしょう。しかし、私のX68030はピンピンしているので、68030だけがおかしくなるという現象はまったく予想していませんでした。

何が悪いんだろうと思っていると、また１人、不具合の報告がありました。

749/999　PED00647まりこ　　　　　RE:040TURBO が来ました
(14)　　93/11/03 06:54　744へのコメント　コメント数：　1

わたしのところでも68030が「ぴ～～～んち」です。症状は
全く同じで「おかしな」の連続攻撃です。
condrv.sysを外すとこの症状がでなくなるのですが，こいつ
は昔からちゃんと動いていたので「はてはて？？？」です。

68040での動作ですが，キャッシュoffだとほとんど問題は（まったくかな？）なくゲームだろうが全部動きます。キャッシュ
onでも「ぱろでぃうすだ！」は動きます。まだ「拡大ふぐ野郎」
まで進めないので効果のほどは・・。ちなみに68030だとここは
相当に滑らかに動くのですね、はい。

750/999　PED00647まりこ　　　　　RE:040TURBO が来ました
(14)　　93/11/03 09:46　749へのコメント

再度，今朝condrv.sysを入れると動くではありませんか！どうも

数時間動かして全体の温度が上昇してくると不安定になるようです。

condrv.sysが変になりはじめるとテキストにゴミが書かれますので
I/O まわりのアクセスタイミングに余裕がなくなるのかも知れません。

これはやっかいです。温度とタイミングの影響は再現が難しいので、仕事でもいつも苦労させられています。発送前のテストで、そういえば１枚バスエラーになった基板があったことを思い出し、あらためてこれをつないで試してみましたが、今度は何も問題が起きません。残っている基板を片っぱしからつないでみましたが、やはり、68030はピンピンしています。熱かクロックの取り出しか、はたまた差し込みが悪いのか、とにかく、見当もつきません。

757/999　PEG00631　BEEPs　re：040TURBO いろいろ
(14)　93/11/03　22:51

こんにちはBEEPsです。

040TURBOが届き始めたようですが、やはり個体差があるのか、いろいろ不具合が出ているようですね。出荷テストのために、全て私の所では動作確認をしてるのですが、環境の違いが大きいようですね。

じゃぎゅあさん＆まりこさん　write

>>　030モードが「ぴーんち」です。

これは困った現象ですね。私の所で68040モード、68030モード共に一応の動作テストしてるのですが、家のは外側カバーかけてないので、熱がこもらず、現象が発生しなかったのかもしれません。

040TURBOの68030モードは、X68030の本来のモードと同じといっても、各信号線の長さは長くなってるし、余分なICが付いてて負荷が重くなってるので、厳密に言えば違いはありますからこれが悪影響してるのかも。
クロックアップしてると、これがモロに引っ掛るようで、クロックダウンしたという報告も聞いてます。

ところで、実は、出荷テストした時、同じように１枚「おかしな命令」が発生してNGになった板がありました。時間がなかったので今までほっておいたのですが、同じ現象だなあと今接続しなおして試してみると、全然その現象が発生しません！

かれこれ3時間になりますが、いたって正常です。condrvも使ってます。

ものは試しで、一度、68030の差し直し、040TURBOの差し直しをしてみてください。直らないようなら、私の所に送り返してください。

BEEPs

764/999　NBH02724じゃぎゅあ　　　030ぴーんち詳細
(14)　　93/11/04　00:28　　　　　　　　コメント数：1

クロックアップはしていないので、伝送ライン長くなり
IC数も増えたのでが過負荷気味になって信号線にゴミ
が載っかっているようです。030モードでウェイト操
作してみると2ウェイト以上にすると問題がありません。
68030と040TURBOの挿し直しは1度しかしていません
が変わりませんでした。
うちのではcondrv.sysでログ参照すると垂直同期異常
になってバスエラーで死んでしまいます。

68030や68040からの発熱は大丈夫です。カバー無し
で、ファンのおかげで良く冷えています。触っても少し
「温かいかな？」くらい。

この状態で2日くらい様子を見てみます。再度挿し直し
とかしてみます。

じゃぎゅあ

ウェイトの挿入[*1]で少し回復するとなると、いよいよタイミングが怪しくなってきます。

しかし、やはり、解決にはならなかったようでした。

＊1
X68030は、X68000相当やX68000 XVI相当の速度にするために、メモリアクセスにウェイトを挿入して実行速度を落とす機能があります。

782/999　NBH02724じゃぎゅあ　　　RE：030　ぴーんち詳細
(14)　　93/11/04　23:53　764へのコメント　コメント数：1

どうもウェイトいれてても「おかしな」でコケてしまう
ようです。挿し直してみましたが相変わらずでした。

```
個体差かもしれませんが、一応送ってみますので
よろしくおねがいします。

デフォルト040になれてしまっただけに、なんか遅い・・・

じゃぎゅあ
```

バグ退治は現象を再現させるのが一番ということで、じゃぎゅあ氏の040 turboが送り返されてきました。同時に、有力な情報も入ってきました。

```
821/999  PED00647まりこ                  RE：030 ぴんち報告
(14)     93/11/06  19:22  819へのコメント  コメント数：1

030モードぴんち！！の結果報告です。結論としてはCRTCアクセス
ではなく，テキストRAMのアクセスでこけることが判明しました。

○ 原因 テキストRAMをポストインクリメントアドレッシングモード
        とDBRA命令で操作するとDBRA命令のフェッチに失敗する

   例1  CONDRV.SYSの場合
          ・・・・・・
          ・・・・・・
        LOOP：
          movem.l d2-d5, -(a0)
          movem.l d2-d5, -(a0)
          movem.l d2-d5, -(a0)   ←ここをnopにするとok!!!
          movem.l d2-d5, -(a0)   ←ここをnopにしても無駄
          dbra d1, LOOP
          ・・・・
          ・・・・
        この場合には，68030の小さなキャシュにも全部コード部分
        が入ってしまいますので  cache on ではコケません。ただ，
        db.x等のようにキャッシュをoffにしてしまうプログラム
        の場合には，見事に暴走します。

   例2  FSX.Xの場合
        LOOP：
            move.l d0, (a0)+
            move.l d0, (a0)+
            ・・・
```

```
. . .
. . .
move.l d0,(a 0)+
dbra d1,LOOP
```

この場合は，ごくたまにコケます。sxwin.xを5～6回起
動を繰り返すとdbraでコケます。

ハード的にはどう信号線をプルアップしても現象はいっこうに改善しま
せんでした。(方法が悪いのかも知れません)

現在，dbraのカウンタ値を2倍に増やしてmove.lの位置にnopを
書き換えして稼働していますが，全く問題無く動きます。
なにしろdbraの命令フェッチに失敗していますから，何が起こって
も不思議ではないのですが，プログラムカウンタがmove.lで操作し
ているアドレス近辺になったり，全く別の位置にブットビしたりで
ハードの不良原因をソフト側から特定できないのが辛いです。

個体差なのか？

じゃぎゅあ氏から送ってもらった040turboを、私のX68030に取り付けて試してみてもいっこうに現象が再現されませんでした。X68030側の個体差の問題のような気がしていましたが、私のX68030で再現しないとなると、ほぼ確実です。

しかし、040turboを取り付けなければ68030の動作に問題ないとなるとX68030のせいにするわけにはいきません。040turboがメモリアクセスのマージンを食いつぶしているのでしょう。

藁にもすがる思いで、040turboのパターンの引き回しをチェックしてみました。

846/999　　PEG00631　BEEPs　　　　　　RE：68030　ぴーんち
(14)　　93/11/07　18:04

いまだ68030ピーンチの原因は不明ですが、040TURBOの基板で
A3の引き回しがちょっと長いというのを発見しました。

可能であれば、68030のA3と連結ソケットのA3を直結して試してもら
えませんか？
A3は、プリント板上の印刷で、D×12の位置です。

BEEPs

じゃぎゅあ氏の基板には、直接、この改造を施して送り返し、再度試してもらいましたが、やはり「68030ぴーんち」は解決しませんでした。

X68030には当たり外れがあります。ハズレを引いたら、040turboの68030モードは使えません。

なんて仕様は、やはり許されないでしょう。これを解決しないかぎり、040turboはあぶなっかしくて配布できません。

ここは徹底的に調べてX68030の不具合をなんとしても発見するんだと決意して、じゃぎゅあ氏にX68030の本体ごと送ってもらいました。

届いたX68030をフロッピーベースでさっそく起動します。これは問題ない[*1]ようです。次にまりこ氏から報告があったcondrv.sysを入れて起動してみます。これまた、ちゃんと起動しました。

ところが、dirでディレクトリを表示させたところ、画面がスクロールしようとした瞬間、

バスエラーが発生しました

おお！

それは、確かに「68030ぴーんち」になるX68030だったのです。ためしに同じ040turboを私のX68030に取り付け、同じフロッピーで起動してみましたが、こちらはまったく問題ありません。スクロールしても平気です。X68030には、まぎれもなく個体差があったのです。

とにかく、現象は目の前に再現できました。後は原因追究です。バスエラーをトリガとして、ロジックアナライザでそこに至るまでの信号線の動きを追っていきました。丸1日かかりましたが、なんとか原因と対処方法を突き止めることができました。

＊1
パターンの引き回しを直したので、少しはよくなっていたのでしょう。

905/999　　PEG00631　BEEPs　　　　　　68030がぴんちの対処
(14)　　　93/11/15　13:45　　　　　　　　　コメント数：4
じゃぎゅあさんとまりこさんのマシンがクロックアップもしてないのに、

「68030ぴ〜んち」になってしまして、いろいろやってみたけど、ラチがあかないので、じゃぎゅあさんから本体を送って貰いまして、調査しました。

結論から言うと、040TURBO接続により信号線の負荷か重くなったことにより発生したものであることは間違いないようです。

不具合の因果関係は、まりこさんが、レポートしてまして、condrv内のテキストメモリのクリアルーチン中で、dbra命令のフェッチを失敗して、変な所にジャンプしてしまうというものでした。

具体的に説明しますと、

```
2CA16    movem.l d2-d5,-(a0)    1
2CA1A    movem.l d2-d5,-(a0)    2
2CA1E    movem.l d2-d5,-(a0)    3
2CA22    movem.l d2-d5,-(a0)    4
2CA26    dbra    d1,$2CA16
```

こういうルーチンなんですが、dbraのジャンプで、$2CA16に行かずに、$2CB16になってました。で、たまたま、ここのコードがSRを操作していて、本来はSRを読み出して'ビットOR'してSRに書き戻しているのですが、途中に飛び込んだために、変な値をSRに書き戻しており、これでユーザバイザモードになってしまい、スーパバイザエリアをアクセスして、バスエラーで落ちてました。

前は，不当な命令になったりして，どこに飛んでいくかわからないという事でしたが，アドレス線A３パッチか　R/W パッチのおかげでか，常にここに飛び込んでいたようでした．

アドレスの変化を追っていくと、68030の命令先読みが効いてか、２番目のmovem後に、２CA24-７番地がフェッチされ３番目のmovemの後に２CA28-B番地がフェッチされてます。この３番目をmovemからnopに変えると、不具合が発生しなくなるとまりこさんがレポートしてましたが、まさにこれがドンピシャで、2CA28番地すなわち、dbraの相対アドレスの読みだしでミスっているようで、ここで、$2CB16にジャンプしてしまうようなアドレスに化けていたのです。

で化けかたですが、本来は、$FFEEにならなければいけない所が、$00EEになってると、こういう現象になります。実際にデータバスの最上位D31-D24をロジアナで当たってみると、たしかに、データの立ち上がりが若干遅い。信号線がバウンドしているようにも見える。オシロで見れば確実でしょうけど、残念ながらないので、これはあきらめ。で、８本とも、みん

な揃って化けるという事は、ゲートが怪しいというわけで、メインメモリとMPUを繋ぐバストランシーバIC 9のゲート信号19番にロジアナのプローブを当てた所、ピタリと現象が出なくなりました。

多分、ゲート信号が反射か何かで安定するのが遅くて、そのため、データバスが開くのが他のビットより遅いのでしょう。それでもギリギリセーフだったものが040TURBOを繋いだことで負荷が重くなって、アウトになってしまったみたいです。まあ他のマシンでは発生しないことから、部品のバラツキなどが原因でしょう。

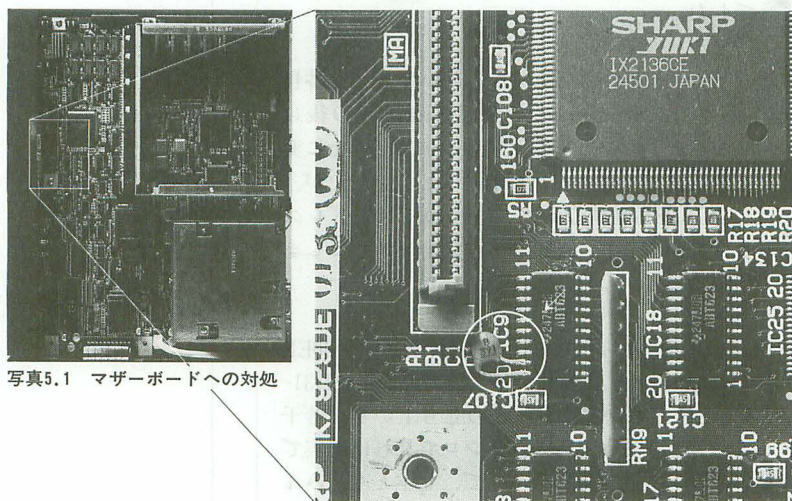
で、対処ですが、プルアップやプルダウンなども試みてみましたが、芳しくなかったため、安易に手持ちのコンデンサ（390pF）をマザーボード上のIC 9 -19とVCCの間にかませてみました。

もともと、この信号はタイミングが厳しいようなので、クロックアップしたマシンで68030側の動きが芳しくない場合も、試してみるといいかもしれません。

BEEPs

じゃぎゅあ氏には、さっそく連絡をとり、X68030のマザーボードへの改造の了解を得て、私の手で対処しました。写真5.1が、その改造結果です。

コンデンサを挿入するのはノイズ取りといった意味がありますが、実際のところ、この波形をオシロスコープで観測したわけではないので、どんな感じに現象が押さえ込まれたのか、本音をいえば、よくわかっていません。とにかく、

写真5.1　マザーボードへの対処

これで収まったからよしとしていますが、根本的には040turboがギリギリのタイミングで動いているのが原因でしょう。

68040側はアクノリッジを変換回路を通して作っているので、ウェイトを入れるなど対処ができますが、68030側は直結しているので、040turboの基板上では小細工ができません。マザーボード上の対処は気持ちのいいものではないですが、とりあえず対処方法がわかっただけでもラッキーです。

```
921/999  NBH02724じゃぎゅあ          RE：68030がぴんちの対処
(14)     93/11/16  06:06  905へのコメント

調査ごくろうさまです。
もしかしてX68030ってギリギリの設計がしてあるとこがあるのか
パターンの引き回しになにかあるのか。

うちの機体の製番は　4311972

じゃぎゅあ
```

```
922/999  PED00647まりこ              RE：68030がぴんちの対処
(14)     93/11/16  18:07  905へのコメント

不安定だった68030モードはBEEPsさんのパッチで（コンデンサは
70pFですが）バッチシ安定動作するようになりました。

もっとも普通は68040で動作させていましたので実害はそれほど
なかったのですが，安心してゲームできます（なんのこっちゃ。）

で，訂正。ネメシスですが68040だとなぜかエナミースローとか
が無限に使えてしまいます。はい。
```

やっかいな問題でしたが、これも隠れていた問題が見つかったということです。じゃぎゅあ氏の協力とまりこ氏の情報がなかったら、もっと深刻な事態になっていたことでしょう。

コピーバックと、またまたチョンボ

コピーバックモードというのは、68040で採用されたキャッシュモードの1つです。68030が持っているキャッシュの場合、メモリ読み込みのときにはキャッシュが働きますが、メモリ書き込みのときには実際にメモリアクセスが起こり、データが書き出されます。

68040でも、デフォルトのキャッシュモードはこの動作です。これを、「ライトスルーモード」と呼んでいます。今までキャッシュオンでやってきたのも、当然、このライトスルーモードでしたが、それでもキャッシュサイズが大きくなったことやバーストアクセスの問題などによって、まともに使えるレベルにするのに苦労しました。

これがコピーバックモードになるとメモリ書き込みにおいてもキャッシュが働くようになるので、データはキャッシュ上でしか更新されず、キャッシュに乗り切らなくなったときだけ適当に書き戻されていきます。*1

このモードだとプロセッサの足を引っ張るメモリアクセスが抑えられるため、高速動作が期待できるわけですが、データを書き込んだつもりでも、すぐにはメモリに反映されないため、キャッシュの制御にライトスルーモードとは違う配慮が必要になります。このため、そもそもコピーバックモードの動作を想定していないHuman68kのIOCSのキャッシュ制御ですから、ちょっとパッチするくらいでコピーバックモードに対応させるのは不可能だろうと思っていました。

コピーバックモードへの対応はバラック基板でHuman68kがまともにキャッシュオンできなかった頃、ベンチマークのためにdb.xを使ってDHRYSTONEを無理やりこのモードにして動作させたとき以来、ずーと、お蔵入りになっていた機能です。

ところが、040turboの参加者の1人からコピーバックモードがどうしてもうまく動かないというメールがきたのです。テストプログラムもついていましたので、さっそく実行してみると見事にバスエラーになってしまいました。

やっぱ、コピーバックは無理だよぉ。

と思いながらテストプログラムの中身を見てみると、透過制御レジスタを使ってキャッシュをコピーバックモードにしたうえで単にメモリ移動を行ってい

＊1
このため、「コピーバックモード」という名前になっています。ちなみに、メモリ上にまだ書き出されていないデータを持っているキャッシュは、特に「ダーティキャッシュ」と呼ばれます。

るだけです。動かないはずはないように思えます。

しかし、実際には何回やってもバスエラーになってしまいます。どこが悪いか見当がつきませんから、db.xでステップ実行させてみると、今度はなかなかエラーになりません。ところが、GOをかけると実行バスエラーになってしまうのです。エラーの場所は、先ほどのステップではエラーにならなかったところです。アドレスが変になっているわけでもありません。さすがに、なんとなく原因の見当がついてきました。

そこで、バスエラーをトリガとしてロジックアナライザで信号線の状態を見てみました。すると、メモリリードが延々と続き、ある程度たまってライトサイクルに切り替わったところでバスエラーになっているのがわかりました。コピーバックモードなので、キャッシュ上でためるだけためて、あふれたところで、書き戻しが起こるのですが、この段階でコケているわけです。

なんでコケるんだ？
コピーバックモードのライトサイクルで何か特殊なことでもしているのか？

実はそうだったのです。

コピーバックは、キャッシュプッシュアクセスという特殊なアクセスモードで実行されるということを思い出しました。これは、68040で追加された機能です。はたと気がついて、68040のアクセスモードを68030のファンクションコードに変換しているIC 4のGALのソースを調べてみました。

前にMMUのテーブルウォークで見た、あのリストです

```
 1:  TABLE tm040 => fc030{
 2:        'b'000 => 'b'111;     <===  キャッシュプッシュアクセス
 3:        'b'001 => 'b'001;
 4:        'b'010 => 'b'010;
 5:        'b'011 => 'b'101;
 6:        'b'100 => 'b'110;
 7:        'b'101 => 'b'101;
 8:        'b'110 => 'b'110;
 9:        'b'111 => 'b'111;     <===  CPU空間
10:    }
```

5行目、6行目がMMUのテーブルウォークでミスっていて直した部分です。しかし、2行目を見ると、まったく同じようにキャッシュプッシュアクセス（000）もまた、CPU 空間（111）に割り付けていました。

確かバラック基板でDHRYSTONEでベンチマークテストを行ったときはコピーバックも動いたゾ。

古いGALのソースを調べてみると、なんとこちらはちゃんとスーパーバイザデータ空間（101）に割り付けられていました。これを、あるときからCPU空間に変更していたのです。いわゆるエンバグというやつですが、その後、コピーバックを試していなかったために気づかなかったのです。何か勘違いしたのでしょうが、やったのはほかならぬ自分ですから、しかたがありません。GALを直してテストプログラムを実行すると、何事もなかったように終了してくれました。

しかし、1匹バグを見つけたらなんとやらではありませんが、MMUのテーブルウォークのミスの箇所から数えてソースファイルでたった3行上のところのミスです。もうちょっとまわりに気を配って見ていればこっちのミスも見つけられていたかもしれませんが、後の祭りです。

もっとも、Human68kがコピーバックモードのキャッシュにまだ対応していませんから、とりあえず68040のコピーバックモードがGALのせいで使えなくても実害はありません。コピーバックモード対応のシステムが出てくるまでにGALを差し替えればいいのです。この時点では、GALの差し替えはまだいつになるかわかりませんでした。

とりあえず「ハードウェアの部屋」で懺悔します。

```
793/999      PEG00631   BEEPs 040TURBOまたもバグ（IC４_ｖ３に！）
(14)         93/11/05   12:18                        コメント数：１

BEEPsです。

昨日の夜、某氏からコピーバックモードが変だというメールがありまして、
調べてみた所、またしてもハードのバグが出てしまいました。

それも、こないだ交換したIC４のMMUのテーブルサーチがCPU空間に
なってたのと同じようなもので、コピーバックもCPU空間になってました。

GALのソースでいうと３行上という「そんなもん一緒に気づけよ」級の
ミスでした（^^;

前回交換のMMUモードのバグも、使わないでいたために発見が遅れたの
```

ですが、GALを作る時、わからん機能はみんなCPU空間にって安易に割り付けてたんで、ぼろぼろです。

それで、またもIC４の処ですが、IC４_Ｖ２の対処でほとんど出払っているし、対処すべきボード数も増えているので一度に差し替えGALを発送する事ができません。IC４_Ｖ２の差し替えで戻ってきた旧GALが来たら、順次回していきますので、すみませんがよろしくお願いします。

まあ、今回のコピーバックモードは、そういうプログラムを組まないと使えない機能ですので、あわてる事はないだろうとタカくくってますが、そういうの使ったプログラム書いててスグ使いたいという人は連絡ください。

BEEPs

コピーバックモードへの対応

テストプログラムとはいえ、コピーバックモードでメモリアクセスが行われているのを見る*1と、無性にコピーバックモードを使ってみたくなります。駄目でもともとと思って、Human68kでコピーバックモードに対応できないかと、040SYSpatch.sysのパッチの検討をしてみました。

キャッシュ制御といっても、キャッシュモードをオフにして動かさなければいけないようなルーチンに関してはライトスルーモードもコピーバックモードもあまり関係はないでしょう。ライトスルーモードでオフにして動かさなければならないようなクリティカルな部分はコピーバックモードでもキャッシュをオフにしておけばいいでしょう。

逆にキャッシュがオンで動いているルーチンに関しては、コピーバックモードだとまずい部分が存在するかもしれません。しかし、具体的に何がひっかかるか予想がつかない*2ので、不具合を見ながら調整していくしかないでしょう。

もう１つ注意しなければならないのは、キャッシュをクリアするタイミングです。ライトスルーモードにおいてキャッシュをクリアしなければならないのは、次のような状況のときです。

１）DMAでメモリにデータを書き込むとき

DMA（Direct Memory Access）*3は、その名のとおり、プロセッサを介在することなくメモリをアクセスすることですから、DMAでメモリにどんなデータが書き込まれたかはプロセッサにはわかりません。このため、プロセッサ内部のキャッシュとメモリ上の最新のデータとの食い違い

＊１
ロジックアナライザでとらえた波形での話ですが。

＊２
実際、通信ソフトで文字落ちするという現象がありました。

＊３
正確には、プロセッサ以外のバスマスタがメモリに書き込むとき、ということになりますが、X68030の場合、DMAC（ダイレクトメモリアクセスコントローラ）くらいしか該当するものがないので、DMAを取り上げています。

が生じている可能性があります。

したがって、DMAが動いた後はキャッシュをクリアする必要が出てきます。

2）プログラムを実行する場合

実行プログラムもフロッピーディスクやハードディスクなどに収められたファイルですから、いったんはデータとしてメモリ上に読み込まれますので、先ほどのDMAの問題を考えなければなりません。さらに、プログラムを実行する時点でもう１つの問題が出てきます。アプリケーションの大半を占める〝.x〟という拡張子を持つ実行ファイルは、プログラムのなかに絶対アドレス形式のメモリアクセスを含むことが許されており、これをプログラムがローディングされるアドレスにあわせてリロケート[*1]してから実行するようになっています。

このリロケート処理はプログラムをデータとみなして書き換えることですから、命令キャッシュには反映されません。このため、プログラムを実行するときにはメモリ上のプログラムと命令キャッシュとの食い違いも問題となり、命令キャッシュをクリアする必要が出てきます。

それでは、コピーバックモードのキャッシュについてはどういう注意が必要になるのでしょうか。コピーバックモードの場合、ライトしたデータはデータキャッシュ上で更新されるだけで実メモリには書き出されません。通常、４セットあるキャッシュラインがいっぱいになると順次書き出しが行われますが、キャッシュ上には書き出しが終わらないデータが存在することになります。この状態でキャッシュをオフにしたりクリアしたりすると、データが失われてし

＊1
この作業のために、〝.x〟形式のファイルにはリロケート情報が付加されています。ちなみに、〝.r〟という拡張子を持つ実行ファイルは絶対アドレスを含まないことになっていますから、リロケート情報は入っていません。ローディングして、そのまま実行されます。

C O L U M N

自己書き換えプログラム

X68030になって「自己書き換えは悪」と決めつけられていますが、命令キャッシュをクリアしてさえいれば、実行時に不具合が生じないようにできます。ただ、680x０の命令セットなら自己書き換えをしなくてもプログラムすることは十分可能です。下手にステップをケチって自己書き換えをして命令キャッシュをクリアするくらいなら、自己書き換えをせずにすますほうがスピード的にも得でしょう。

なお、やむを得ずに自己書き換えを行う場合でも、プロセッサのキャッシュクリアの命令を直接使わず、IOCS-ACを使ってクリアするようにしましょう。

まいます。これを避けるためには、強制的にキャッシュ上のデータを書き出してからキャッシュをクリアしなければなりません。このための専用の命令「cpush」が68040には用意されています。

しかし、ライトスルーモードで使っていたキャッシュクリア命令「cinv」のかわりに、コピーバックモードではこのキャッシュプッシュとクリアを行う命令「cpush」に置き換えるだけでいいかというと、そう簡単にはいきません。コピーバックモードは、ライトスルーモードより状況がこみいっているため、次のような点を注意しなければなりません。

3）DMAがメモリのデータを読み込むとき

ライトスルーモードでは、DMAのメモリ書き込み時のみを問題にすればよかったのですが、コピーバックモードではDMAの読み込み動作のときも問題になります。メモリに書き戻されていないデータはDMAからはアクセスできないので、メモリ上の古いデータを読み込んでしまうからです。このため、DMAが動く前にデータキャッシュをプッシュしておく必要があります。

4）DMAがメモリにデータを書き込むとき

ライトスルーモードでは、DMAの動作後データキャッシュをクリアするだけでよかったのですが、コピーバックモードの場合はそうはいきません。DMAの動作後にキャッシュプッシュすると、せっかくDMAが書き込んだデータをキャッシュプッシュで上書きしてしまう可能性があります。

C O L U M N

バススヌープ

68040は、DMAなど他のバスマスタが動いてもキャッシュとメモリが一致するようにするための「バススヌープ」というハードウェア機能を持っています。これは、他のバスマスタが動いているときにバスを監視して、メモリリードがダーティキャッシュに対するものだった場合、メモリのかわりに68040のキャッシュからデータを送出したり、メモリライトのデータをキャッシュに取り込んだりすることを実現するものです。

しかし、このバススヌープ機能を活用するためには、メモリシステムやバスマスタがそれに対応していなければなりません。X68030のメモリシステムは、当然そんな対応はしていないので、これはあきらめています。このため、「cpush」によりソフトウェア的にキャッシュとメモリの内容を一致させることになります。

DMAが書き込んだ範囲を避けてキャッシュプッシュすればいいのですが、IOCSのルーチンは一括してクリアするようにしかなっていません。このため、この対処もDMAが動く前にデータを書き出してからキャッシュをクリアすることになります。結局のところ、意図するところはちょっと違いますが、対処方法としてはDMAの読み込みのときも書き込みのときも、同じようにDMAが動く前に「cpush」すればよいことがわかりました。

5）プログラムを実行する場合

ライトスルーモードでは命令キャッシュをクリアするだけでよかったのですが、あらためて命令フェッチを開始しても、コピーバックモードではプログラムとして読み込むべきメモリ上のデータがまだデータキャッシュから書き戻されていない可能性があります。このため、プログラムの実行を開始する前には、命令キャッシュのクリアだけでなく、データキャッシュのプッシュもしなければなりません。

これらの検討を踏まえたうえでIOCSやHuman68kのキャッシュ制御部分を調べていくと、実はIOCSもHuman68kも大雑把な制御しかしていないことがわかってきました。

フロッピーディスクおよびハードディスク系のIOCSコールでは読み込み、書き込みを問わず、また、DMAの起動前や後などには頻繁にキャッシュをクリアするサブルーチンを呼び出していました。このサブルーチンは、一律にすべての命令キャッシュとデータキャッシュをクリアするだけのものですが、ADPCMの関係も含め、IOCSルーチンはほとんどがこのルーチンを呼び出しています。

このため、ライトスルーモードとコピーバックモードの違いで問題となるDMAの書き込みと読み込みの違いや、起動の前と後の違いなどについては、いちいち考える必要がなさそうです。簡単にはすみそうにないと思っていたコピーバックモードへの対応でしたが、IOCSのほうは「cinv」を使ってキャッシュクリアしていたところを、「cpush」に書き換えるくらいですみそうです。

一方、Human68kのほうは次に挙げる箇所でキャッシュをクリアしていますが、IOCSのように命令キャッシュとデータキャッシュを一律にクリアするのではなく、目的にあわせてクリアするようになっています。

(a) 〝move from SR〟を〝move from CR〟に書き換えている部分で命令キャッシュをクリアする。

(b) プログラム実行の開始前に命令キャッシュをクリアする。
(c) デバイスドライバの呼び出しの部分でデータキャッシュをオフにし、デバイスドライバから戻ってきたところでキャッシュをクリアする。

(a)、(b)に関しては、検討項目で説明したように、命令キャッシュをクリアしてもコピーバックモードのデータキャッシュが働いているとメモリ上に書き戻されていない状況が起こるので、命令キャッシュと同時にデータキャッシュのプッシュが必要になります。

また、(c)に関しては、キャッシュのオフと同時にデータキャッシュをプッシュすればよいのです。

コピーバック対応040SYSpatch ver2.1a

これらの問題にあわせて、ライトスルーキャッシュを動かすために040SYSpatch.sysで行っていたパッチ部分をコピーバックモードで不都合がないように直してみると、絶対無理だろうと思っていたコピーバックモードも意外に簡単[*1]に使えるようになってしまいました。

案ずるより産むがやすし、ということでしょうか。もっとも、ライトスルーモードでキャッシュオンにできるようにするためにいろいろ解析してあったのでなんとかなったのであって、いきなりコピーバックモードでやろうとしたら無理だったかもしれません。

約束のDHRYSTONEもちゃんと動いて、値は25000を超えました。前にバラック基板で実行したときの数字とほぼ同じですから、ちゃんとコピーバックモードで動いているといえるでしょう。もっとも、プログラムによっては「おかしな命令を実行しました」といってきたり、「バスエラー」が起こったりしますので、まともにコピーバックに対応したというよりはHuman68kのシステムが暴走しないで動くだけは動いているというレベルです。

また、もともとライトスルーモードを想定したHuman68kを、コピーバックモードで矛盾が起きないように無理やりパッチを当てているだけですから、見た目は暴走しないで動いていてもどんな問題が隠れているかわかりません。

不安定なプログラムを公開するのは気が引けますが、1人で使うよりは多くの人に試してもらったほうがバグ出しも早いし、もともと、今回の基板配布はバグ出しの協力者を募るという意味もあったので、多少問題があってもプログラムはどんどん公開して試してもらいます。

テストバージョンと断ってコピーバックモード対応版、040SYSpatch ver2.1aをアップロードしました。

＊1
本当は、試行錯誤でパッチを直していったのです。ちょっとやってみて暴走させては原因を考えて、ということの繰り返しでした。

928/999　PEG00631　BEEPs　040SYSpatch ver2.1a コピーバック対応です
(14)　93/11/18　00:33　　　　　　　　　　コメント数：1

BEEPsです。

junk-shopに040SYSpatch ver2.1a を放り込んできました。
68040のコピーバックモードに対応した040SYSpatchです。

といっても、従来のライトスルー対応のパッチの延長で、コピーバックだと問題になりそうなコード書き換え後のキャッシュ制御について、取りあえず対処したというだけで、不具合がまだいっぱいあります。

不具合が、そもそも自己書き換えなどの問題によるものなのか、パッチプログラムのバグなのか、パッチしたHumanがコピーバックに対応できていないのかさえわかってません。

もともと、Humanがコピーバックモードなあんて考慮されてるハズないんで、ちょこちょこっとパッチ当てるだけで対応できるとは思ってませんでしたが、全然だめだと思ってたわりには、意外とそれなりに動いたんで、取りあえず公開しました。

感じとしては、ライトスルーの2倍はちょっと手がとどかないかなってところ。

お約束の
ドライストンは、25000.0（ver2.1の500000回)
ウェットストンは、2564.10
gcc.x,emacs.xはサクサクで動いてます。

でも、sxは全然ダメですね。起動すらしません。[*1]

あと、lzxがかかってるとバスエラーになりますね。patexecがキャッシュフラッシュしてくれてると思ってるのだけど。

ついでに、何でかわかりませんが、040SYSpatchの登録が2回行なわれようとします。これまた謎。

というわけで、ちょっとこいつは常用するというわけにはいきませんが、取りあえずαテストという事で使える環境にある人は試してみてください。

なお、IC4-V3でないと、使えません。

＊1
今はパッチをすればコピーバックモードでも問題なく動くようになりました。

BEEPs

PS.　11/１以前に発送した040TURBOに搭載されているIC４-Ｖ２は、キャッシュオンでバスエラーになります。IC４-Ｖ３が届くまで今しばらくお待ちください。

しかし、プログラムを公開したのはいいですが、コピーバックモードのバグを抱えたIC４のままでは、これは実行できません。Human68kでは永久にコピーバックモードは無理だろうと悠長にかまえていたために、バグ取りしたGALの発送作業をサボっていて、実はほとんどの人は試したくても、試せないという状況でした。

こんなことなら、もっと早くからGAL交換作業をしとくんだった。

自分で自分の首を締めているなあと思いながらも、あわててGALをかき集め、差し替え用のIC４-Ｖ３の発送作業を始めました。

cache.xでエラー

コピーバックモードでは何が問題になるか、一例としてcache.xの問題について紹介しておきましょう。

cache.xは、X68030ユーザーにはおなじみの、68030の内蔵キャッシュをオン・オフするためのプログラムです。このプログラムはIOCS-$ACを呼び出してキャッシュの制御をしているので、IOCS側のルーチンを68040対応にするだけで、cache.x自体はパッチしなくても68040の内蔵キャッシュも同じように制御することが可能でした。

ところが、コピーバックモードでcache.xを使うと、キャッシュをオンにするときはいいのですが、キャッシュをオフにするとき、「おかしな命令を実行しました」とか、「バスエラーが発生しました」といったエラーが発生してしまいます。

しかし、再度cache.xを実行すると、ちゃんとキャッシュはオフになっているのです。いちおう、目的とすることは達成できているのですが、なぜかエラーが発生するのです。発生している現象だけを見ると、「68030ぴーんち」のときの状況に似ていなくもないので、最初は、

コピーバックでどこかメモリアクセスがクリティカルになっている部分があ

るのか？

と考えました。

デバッガのステップ動作で追いかけていくと問題なく動いてしまいますので、これまたやっかいです。いろいろ条件を設けて原因を絞っていくと、IOCS-$ACを呼んでキャッシュ制御するまでは正しく動いていて、キャッシュをオンからオフにした後のどこかで変な番地に飛んで行って暴走してしまうことがわかってきました。どこで飛ぶか、どこに飛ぶかは実行するときによって変わります。原因は純粋にソフトウェアの問題だったのですが、「68030ぴーんち」のことが頭にあったので、なかなかほかのことに目が向きません。ハードウェアのタイミングを疑ったりしてずいぶん無駄な時間を費やしてしまいましたが、コピーバックモードの動作を冷静に考えればわかることでした。

IOCS-$ACは、キャッシュの状態読み出し、オン・オフ、クリアなどのサービスルーチンに分かれていますが、問題はこのうちのキャッシュをオンからオフにするときです。IOCS-$ACのキャッシュをオン・オフしているルーチンは、キャッシュ制御レジスタのなかにある命令キャッシュとデータキャッシュのオン・オフを制御するビットを操作しているだけです。コピーバックモードでは、キャッシュをオフにしたとき、キャッシュプッシュもいっしょに行わなければなりませんが、IOCS-$ACのキャッシュクリアルーチンは別になっているために見落としていたのです。

このため、いきなりキャッシュがオフにされてメモリに書き戻されないままデータがキャッシュ上に宙ぶらりんで残ってしまい、メモリ上の古いデータをアクセスしてしまうのです。今回の場合は、サブルーチンコールで戻りアドレスがスタック上に書き込まれるはずのところが、実際にはキャッシュ上に書かれただけだったため、リターン時にスタック上の古いデータをアクセスして、変な番地に戻って暴走していたわけです。

040SYSpatch.sysでIOCSルーチンのキャッシュ制御とともにキャッシュプッシュするようにしたところ、cache.xの不具合は収まりました。[*1]

＊1
実はこのときの処置はまだ不十分で、コピーバックにはまだ問題が隠れていたのですが、動くようになったからOKと思って、それ以上のことには頭が回りませんでした。これについては、後ほど説明します。

コピーバックとライトスルーの混在

コピーバックモードを扱うためのノウハウがわかってくると、エディタやコンパイラなど、自分が使っている主なプログラムはなんとか使えるようになってきました。

lzx化していたプログラムは軒並みおかしくなってしまいましたが、lzxの自己書き換えという性質上、しかたがありません。lzxを解除*1して使うことにします。

ほかにはプログラム中で独自に自己書き換えを行っているプログラムや、DMAを直接使っているようなアプリケーションで問題*2が出ました。

また、ちょっと予想していなかった問題として、通信ソフトで文字落ちを起こすという不具合も見つかりました。これは、コピーバックモードでダーティキャッシュがたまっているところにcpushで強制書き戻しをかけると、すべての書き込みが完了するまで割り込みなどの他の処理を受け付けられなくなるからのようでした。これは、アプリケーション側で回避するのは難しい問題です。

こうした不具合の出るアプリケーションでも、キャッシュをオフにすれば、たいていのプログラムは実行することができますから致命的ではないのですが、040turboの場合、68040のキャッシュオフでの動作は、68030のキャッシュオフでの動作よりも遅いのです。このため、68040のキャッシュオンとキャッシュオフの落差は68030よりもはるかに大きく、キャッシュオンのスピードに慣れてしまうと、キャッシュオフの動作の遅さが耐えがたいものに感じられます。

すべてがコピーバックモードというのは無理にしても、ライトスルーモードでいいからキャッシュをオンにして動かしたいところです。ライトスルー版の040SYSpatch.sys ver2.0とコピーバック版のver2.1aを、いちいち入れ換えていたのでは面倒ですから、キャッシュをオン・オフするのと同じようにキャッシュモードのほうもコピーバックとライトスルーを簡単に切り替えられると便利です。

考えてみると、040SYSpatch.sys自体はMMUによりページごとにキャッシュ領域と非キャッシュ領域を設定するようになっており、ver2.0ではキャッシュ領域をライトスルーモードに、ver2.1aではコピーバックモードになるように初期化しているだけですから、これを自由に設定できるようにすればよさそうです。

こうして、ページ単位にキャッシュモードを指定できる機能*3をサポートし

＊1
現在は、中村ちゃぷに氏作のダイナミックパッチャpatexec.sysがコピーバックモードに対応するようになったので、lzx化されているプログラムでも平気になったようです。

＊2
これらのプログラムは68030でも不具合が出る可能性を持っているのですが、68030のキャッシュがライトスルーモードであり、そのうえ、キャッシュ容量が小さいために問題が表面化しない場合があります。

＊3
この機能は、IOCS-$ACの機能拡張の形で実装しました。

たver2.1cができあがりました。

さらに、実際のプログラムを実行する際にいちいちMMUのページを指定してキャッシュモードを変更するのも面倒なので、フリーエリアをまるごと指定したキャッシュモードに設定する040cache.xというプログラムを用意しました。

実際の使用では次のように簡単な操作でキャッシュのコントロールができるようになります。

```
cache on
```

これでキャッシュがオン*1になります。この状態では、まだライトスルーモードです。

*1
040cache.xは、あくまでキャッシュモードを切り替えるだけなので、cache.xでオン・オフをしなければなりません。

```
040cache c
```

これでコピーバックモードになります。後は実行したいアプリケーションを走らせれば、コピーバックモードの状態で走ります。

コピーバックモードで不具合が出るようなら、次のようにしてライトスルーモードに戻します。

```
040cache w
```

COLUMN

キャッシュ制御コマンド

現在では、040cache.xに加え、参加者の手による次のようなプログラムが開発されています。

allcache.x じゃぎゅあ氏作

040cache.xはフリーエリアだけでしたが、こちらはメインメモリの全領域を対象としてキャッシュモードを指定します。非常に高速になります。

setcache.x Y.氏作

040cache.xの機能に、アドレスをじかに指定することができるようにしたものです。より細かい設定ができ、玄人向けです。

これでライトスルーモードになりますから、あらためてアプリケーションを走らせます。

これでも駄目なら、これが最後の手段です。

```
cache off
```

キャッシュをオフにして、実行します。

これで、ほとんどの場合、問題なく使用できるようになりました。

コピーバックモードとライトスルーモードのパフォーマンス

さて、期待のコピーバックモードのパフォーマンスを見るため、いろいろな方法でメモリアクセスして調べたのが、表5.1です。2行目で256バイトおきにアクセスすると68030がキャッシュオフ並みの性能しか出ないのは、ダイレクトマップ方式のキャッシュの宿命です。

さて、ライトスルーとコピーバックの比較ですが、これは3行目を見ると明らかです。ライトスルーではライト時にキャッシュオフのメモリアクセスと同じ時間がかかるわけですから、キャッシュオフとコピーバックキャッシュのちょうど中間の値となるわけです。もっとも、普通のアプリケーションではリードと同じ頻度でライトすることはないのですから、ここまで顕著な差は出ないでしょう。

一方、4行目はリードの後にファイルアクセスをしたものですが、この性能はキャッシュオフ並みになっています。これは、システムがファイルアクセス時に実際にキャッシュをクリアするようになっているためです。また、5行目ではキャッシュオフよりも性能が落ちてしまっていますが、これは、68040の場合、キャッシュオン時には必ずライン単位でメモリアクセスするため、無駄なアクセスが増えてしまうからです。

表5.1　キャッシュモードとメモリアクセス時間（単位nsec）

プロセッサ	68030		68040		
データキャッシュ*1モード	off	on	off	w-on*2	c-on*3
move.lを4個並べて連続リード	1410	970	950	120	120
256バイトおきにmove.lを4個	1420	1420	950	120	110
move.lのリードとライトを各2個	1210	970	920	530	110
move.lを4個並べて連続リード後ファイルアクセス*4	2850	2920	2610	2560	2560
256バイトおきにリード後ファイルアクセス	2850	2920	2610	4960	4950
4レジスタを使ってmovem.l	1250	1210	1070	240	230

＊1　命令キャッシュはオン。

＊2　ライトスルー。

＊3　コピーバック。

＊4　ファイルアクセスの時間含まず。

逆に、いったんデータをメモリ上に持ってきたら、後はひたすら自分の内部で処理を進めるというタイプのアプリケーション[*1]の場合は効果が期待できます。

それにしても、フロッピーディスクやハードディスクなど、DMAを使っている場合はあきらめもつきますが、RAMディスクのようにソフトウェアのメモリ間の転送だけの場合はキャッシュをクリアしてしまうのはもったいないところです。これは、Human68kがデバイスドライバを呼び出すときに必ずデータキャッシュをクリアしてしまうのが根本原因です。デバイスドライバ側でキャッシュを必要に応じてオフしたりクリアしたりするなどの配慮をしていればいいのですが、X68000時代に作られたプログラムをそのまま使っても問題が出ないようにするために、デバイスドライバの呼び出し側でキャッシュをクリア[*2]しているのでしょう。

さらに、IOCS内のルーチンはあちこちで過剰とも思えるくらいキャッシュのクリアをかけています。こちらは、なお悪いことに、命令キャッシュもデータキャッシュもみんな丸ごと捨ててしまっています。まあ、このおかげでX68000時代のかなりトリッキーなプログラムであっても結構動いているのでしょうが、ちょっとやりすぎのように思います。必要な範囲だけにかぎって[*3]キャッシュをクリアするようになっていればいいのですが、そうなってはいません。

クリアしなければならない範囲がわかれば、部分的なクリア[*4]に改造することもできるのですが、残念ながら、IOCS内でキャッシュクリアを一手に引き受けているルーチンには、このキャッシュクリアの範囲に関する情報は伝わってきません。キャッシュクリアを呼び出している側のルーチンを解析して、できるだけ効率の落ちないようにキャッシュをクリアするのが理想ですが、現状では、そこまで手が回らないため、68030と同じようにまとめて全部クリアするしかできません。

68030なら命令キャッシュ、データキャッシュあわせても、たかだか512バイトしかないので、丸ごと捨ててもたいした無駄ではないのかもしれませんが、68040は8Kバイトが丸ごと捨てられてしまいますから、実に16倍です。それに、キャッシュプッシュを行う「CPUSHA」命令は、なんと、最低でも数百クロックかかり、メモリへの書き戻しの時間も含めると、数千クロックという、とてつもないサイクルを消費します。

この間、割り込みを受け付けることもできないので、通信ソフトで文字落ちを起こすという問題[*5]を引き起こしています。

＊1
レイトレーシングなどのプログラムが、これにあたります。かつて68030＋68882の組み合わせの凄さを実感した例のレイトレーシングプログラムは、68030と比べライトスルーモードで約2倍、コピーバックモードでは約3倍になります。

＊2
最新版の040SYSpatch.sysでは、オプションでこれをスキップするようになっています。

＊3
68030の場合、キャッシュを部分的にクリアするにはCAARレジスタを使っていちいちアドレスを指定しなければなりません。これは面倒なので、一括してクリアしているのでしょう。

＊4
68040の場合、ページ単位でクリアする命令があるのです。

＊5
ライトスルーモードで実行すれば大丈夫です。

なんとか無駄なキャッシュクリアを減らして、効率よく実行できるようにしたいというのが、今の課題となっています。

第6章

第二次配布

今度の基板はバグってる?

第一次配布が始まり、040turboの話が盛んになると、噂を聞きつけて多くの人が問い合わせをしてきました。もともとは第一次配布の参加者からの反応の様子を見てハードウェアに改良を加えることを考えていたので、すぐに第二次配布をするつもりはなかったのですが、ハードウェアの調子は予想よりもいいものでした。

このため、ちょっとばかり自信をつけて1993年の11月早々に第二次配布希望者を募ることに踏み切ることにしました。第一次配布で結構な数の68040の余りが出たので、これを浮かせておくのがもったいないというのも、第二次配布を早める理由の1つになっています。

なぜ余ったかというと、040turboの配布を始めた頃、ジャンク品の68040を持っていたPA 5氏が話を聞きつけて安価に譲ってもよいと連絡してきてくれました。このため、ジャンク品でも安いほうがいいという人には68040を抜いた基板のみを配布することに変更したので、手配した68040が余ったのです。もちろん、このジャンク品、ちょっとチップのマスク版数は古いものでしたが、問題なく動きました。

COLUMN

ジャンク品の68040

X68030のプロセッサはMMUを持たない68EC030ですが、これをMMU内蔵のフルスペック68030に交換している人は結構多いようです。Human68kで使っている分にはMMUは活用されないので68EC030でもかまわないのですが、この68030に交換すると、起動時のIPL画面で〝On-Chip-MMU〟の表示が出るようになります。

秋葉原のジャンクショップですと68030のジャンク品は1万円以下で売っていますから、これを購入しても、そう高い買い物ではありません。

私もそうですが、たいていの人はこのジャンク品の値段のイメージがあるので、68030は安いもの、68040は高いもの、と思ってしまいますが、68030も新品は68040並みの値段がするのです。最近では、MacintoshのCentrisやLC475、LC575など、68LC040を搭載した機種向けに68040を売っているショップなどがあるので、68040を個人で買うことも比較的簡単になりました。そろそろジャンク品の安い68040も出回ってこないかなと思っています。

さて、自信を持って臨んだ第二次配布でしたが、納入された基板をテストしてみると、起動途中でハングアップしたり、バスエラーになったりする物が続出してしまいました。

第二次配布の基板は、「68030ぴーんち」を調査しているときに気になったプリント基板の2本のパターンについては改造を加えて引き回しを変えてみましたが、回路的には第一次配布とほとんど変わっていません。改造が悪かったのかとも思いましたが、前の基板を改造したときはこんなことにはなりませんでしたし、68040のチップのマスクが変わったのかとも思いましたが、それもありませんでした。簡単には解決しそうになかったので、とりあえず、動く基板だけをテストして発送作業を行いました。

製造不良を疑って業者に送り返そうかと思っていたところ、実はパッチプログラム040SYSpatch.sysが犯人でした。

第二次配布記念040SYSpatch ver2.2

コピーバックとライトスルー両モード対応版である040SYSpatch ver2.1cによって、68040の目玉となる機能のサポートがひととおりできるようになりました。その後、じゃぎゅあ氏の手によるROMデバッガへの対応や、PUNA氏の手によるHuman68k ver3.02対応などの機能が盛り込まれて、040SYSpatch.sysは最終的にver2.1fまでバージョンアップし、細かい不具合はあるものの、機能的にはほぼ問題ないものになりました。

細かい不具合といっても、IPL画面でのクロックの表示が文字化けしてしまうとか、SRAMに設定したキャッシュモードが起動時に反映されないとかいったものです。いわゆるKnownバグ[*1]で、実害がないので、そのままになっていたのでした。

せっかく第二次配布も始まることだし、040SYSpatch.sysも安定してきたので、ここらへんの細かい不具合も直しておこうと作ったのがver2.2でした。

＊1
発見されて、なお、生き残ってるバグです。対応が面倒もしくは直してもあまりメリットがないので、後回しになっているというのが普通です。ちなみに、謙虚さを忘れたソフトウェアメーカーにいわせれば「仕様」になります。

149/299　PEG00631　BEEPs 040turboの第二次分と、SYSpatch ver2.2
(3)　　94/01/22　01：11　　　　　　　コメント数：　2

BEEPs です。

第二次配布分の基板は、今日(1/21) 届く予定でしたが、発送が若干遅れたそうで、明日(1/22　もう今日ですが)の午前中に届く予定とのことです。
今日、出荷テストのために、早く帰ってきたのですが．．．．

明日、出荷テストが順調に行ったら、日曜日には届くかもしれません。

ところで、040TURBOで起動する時、IPL画面のクロック表示がおかしくなっていたのですが、これは、キャッシュをオンにしてクロックの計測ルーチンに入ると、ハングアップしてたので、キャッシュをオフにしてたからです。でも、今回やってみたら、まともに動いてしまいました。68040のマスク版数の問題なのかもしれません。

この、クロック表示ほか、いろいろ姑息な修正をした040SYSpatch.sys ver2.2を、Junk-Shopに入れておきます。
二重に登録されてしまうバグ等は潰しましたが、機能的には、ver2.1fとほとんど変わってません。

お試しください。

また、オプション'%'のクロック表示がうまく動くかどうか、ご報告ください。

BEEPs

さっそく答えが返ってきました。

150/172　HGE02300　TeM　　　クロック表示でます　SYSpatch ver2.2
(3)　　94/01/22　02：07　149　へのコメント　コメント数：　1

ども、TeMです。

早速、　040SYSpatch.sys ver2.2　をいただきました。
取りあえず、組みこんだところちゃんとクロックが表示されました。

》てみたら、まともに動いてしまいました。68040のマスク版数の問題な
》のかもしれません。

ちなみに、私の040TURBOは68040込みの構成でしたので、BEEPsさんのマスクと一緒でしょう。

第二次配布分、頑張ってください。

どうも、ありがとうございました。

PS. クロック表示値は　25.1MHz でした。　(^^;

TeM

153/172　KHF03720　なっち　クロック表示OK　(040SYSpatch v2.2)
(３)　94/01/22　21：01　149　へのコメント　コメント数：　1

なっちです。

040SYSpatch.sys ver2.2組み込みました。僕のところでも無事クロック表示が
出るようになりました(TeMさんと同じく25.1MHz)。ちなみに、僕の040はPA５さんからゆずってもらったものです。
MMUやFPUの表示も出ました。
ただ、ドキュメントはEUCで書かれていました。　(^_^;)

今まではver2.1eを使っていたんですが、これだと起動時に必ずキャッシュがOFFになっていたんですが、ver2.2ではキャッシュは起動時にはちゃんとSRAMの設定値に設定されるようになりました。(もしかして2.1fからすでにそうなっていたのかもしれません。　2.1f は使っていないものですから…。)

「姑息な修正」なんて書かれていますが、これらの表示がきちんと行われるととても気持ち良いですね。気分的には「大幅な改良」に思えます。

ROMVER2.Xという起動時の表示と同じものを出力するソフトを持っているんですが、これを実行してもちゃんとクロック表示とFPUの表示が出ました。(MMUの表示は出ませんが、IOCS-ACで得られる情報はMMU無しのままだということですから、これも正常でしょう。)

なっち

ver2.1fからver2.2への修正はたいしたことでもないし、みんなもOKということなので、大丈夫だろうと簡単に考えて、第二次配布の基板もこの新バージョンでテストしていたのですが、実はこれがエラーの原因だったのです。

第一次配布の基板では動いたということだから、第二次配布の基板の問題と思ったわけですが、一夜明けたら第一次配布の基板でも、やっぱり不具合の報告が上がってきました。

154/172　HGE02300　TeM　クロック表示でます。と思ったら．．？
(3)　　94/01/22　21：10　152　へのコメント

TeMです。

040SYSpatch.sys ver2.2　なんですが、その後一度電源を落として再起動したら、バスエラーが出るようになってしまいました。　(T T)

で、色々試したのですが、まず以前のバージョンの　040SYSpatch.sys を組みこんで立ち上げた後、CONFIG.SYS をver2.2の040SYSpatch.sys に書き換えて立ち上げ直すと、バスエラーが出ずに立ち上がります。

HUMAN.SYS は3.01　3.02とも試してみました。
040SYSpatch.s をアセンブルし直しても、同じでした。

何か、間違っているのでしょうか。

TeM

エンバグか？

第一次配布の人からも起動時のバスエラーの報告があったということは040SYSpatch.sysのバージョンアップの際のエンバグの線が濃厚です。もう一度、エラーになっていた基板をセットして起動してみると、エラーになったり、ならなかったりします。とっかえひっかえしていくうちに、1枚、いつでも、必ずバスエラーになる基板が見つかりました。バスエラーになるアドレスは$02F3F0、これは040SYSpatch.sysの中です。

実際にはどの部分なのか、このアドレスを控えておいて68030モードで起動してみます。68030モードではパッチの処理はスキップしますが、040SYSpatch.sys自体はデバイスドライバとして登録されます。メモリ上のデバイスドライバをサーチして040SYSpatchの登録された先頭アドレスを調べてみると、

$02F096でした。先のバスエラーのアドレスから040SYSpatchの先頭アドレスを引くと、差は$00035A、ここまでわかれば、しめたものです。040SYSpatch.sysをデバッガで読み込んで、先頭からこの差にあたる$00035Aの位置で何をやっているかを調べてみると、プロセッサが68040かどうかを判定しているルーチン

```
CPUSHA IC/DC
```

この命令を実行している場所でした。

これはキャッシュプッシュ命令で、68040の場合は実行できますが、68030にはこの命令が存在しないので、未定義命令*1になります。

これをつかまえてプロセッサの種類を判定しているわけです。そして、これがバスエラーを引き起こした張本人でした。

68040はリセットされるとキャッシュ機能をオフにしますが、キャッシュラインの中身のクリアは保証されていません。このため、68040のマニュアルでは、最初にcinv命令を使って明示的にキャッシュをクリアすることと明記されています。040SYSpatch.sysがライトスルーモードしかサポートしていなかったときはキャッシュクリアをすべてcinvでやっていたのですが、コピーバックモードに対応するためにcpushに書き換えた際に、このプロセッサの判定部分も書き換えていたのです。このために、電源投入時、キャッシュラインにたまたまダーティキャッシュ状態を示すビットが立っていると、cpushによって、このデータを書き戻そうとしてしまいます。当然、このキャッシュラインは、データはおろかアドレスタグさえ、でたらめです。変なアドレスを指していればバスエラーになりますし、たとえメインメモリを指していてバスエラーにならなくても、プログラムが破壊される可能性があるのです。エラーになったり、ならなかったりというのは、これが理由でした。

この部分をcinvに直してみたところ、起動時のバスエラーはなくなりましたので、とりあえず、この部分だけ直して040SYSpatch.sys ver2.2aを公開しました。

しかし、よく調べてみると、これはコピーバックモード対応になったver2.1aのときにエンバグしていたのです。なぜ、今まではバスエラーにならなかったのか、そして、なぜ、ver2.2になっていきなり出るようになったのか、これはよくわかりません。

隠れていたバグが1つ見つかったからよしとしよう。

＊1
本当は、このキャッシュプッシュ命令のマシン語コードは$F4F8なので、68030ではFライン命令になります。だから、プログラム自体が間違っているのですが、それでも運よく動いていました。ちなみに、このバグ、じゃぎゅあ氏により発見されました。

そう思っていたのですが、実は1つではすまなかったのです。

まだまだ続く問題点

さて、これで直ったなとひと安心していたのですが、そうは甘くはありませんでした。コピーバックモードで動かなくなるプログラムが続出したのです。

167/172　PAF03012　Arimac　040SYSpatch 新版で…
(3)　94/01/24　00：48　コメント数：　2

　040SYSpatch.sysなんですが、家ではVer2.2とVer2.2aの両方とも
PIC, DJ, JPEGED, HAPICを実行すると「おかしな命令を実行しました」
や「アドレスエラー」、「CHK命令を実行しました」等のエラーが
出るようになりました。

　これらは、データキャッシュがOFFになっているか、ライトスルー
にすると出なくなります。

　私のところでは　040cache.s をいじって、アドレスの切り捨て、
切り上げを逆にしてコピーバックの範囲を広げてあるのですが、
これだと元の040cacheに比べて非常に顕著にこの現象が起きます。
もちろん、元の040cacheでも起きます。

　原因は私ももう少し調べてみたいと思いますが、取り合えず
ご報告しておきます。

　一応、私の　040TURBO のデータ
・　68040：PA 5さんから譲って頂いた物
・　GAL : IC 4 _V 3

Arimac

179/183　MXC02770　Y.　RE：040SYSpatch 新版で…
(3)　94/01/25　21：03　175　へのコメント　コメント数：　1

私も、040_22aSyspatchで、arimacさんとほぼ同様の症状がでています。

確認しているのは、miki.x,TMN Σ（オリジナル、をたく版とも）FSX.
X（コマンドラインから起動、040用パッチずみ）と、バイナリファイル

が大きいものです。(GCC EM.Xは問題ありません。また、いずれも、040_21dSYSpatchでは問題無く動いていました。)
Humanは3.01
040sysはパラメータなしで組み込み
sramのキャッシュ設定は共にOFF
autoexec.batの最後でcache on, 040cache cを実行しています。
(ライトスルーなら問題なし)

040TURBOは一時配付分ですから、BEEPsさんのと同じでしょう。

立ち上げると白い窓がでて、'A'中止でコマンドラインに戻ってくれば、
再度立ち上げると、なにもなかったように動き出します。

FDにコピーしたTMNを立ち上げても同様です。

Y.

182/183　NBH02724　じゃぎゅあ　RE：040SYSpatch 新版で…
(3)　94/01/26　00：05　179　へのコメント

私も同じような感じでコピーバックでハングってたのでtmn.xの
起動で試してみました。

組合せHuman68k＋040SYSpatch.sys＋PATEXEC.SYS v0.2

1. v3.01　v2.1f
2. v3.02　v2.1f
3. v3.01　v2.2a
4. v3.02　v2.2a

1.
正常。全く問題無く起動する。

2.
A（中止）のみ選択可能だが何度やっても抜けられないハング
アップ状態。

3.
A（中止）のみ選択可能で異常終了して、データキャッシュが
オフになっている。
コピーバックモードではデータキャッシュオフでも、異常が
頻繁に発生しアボートする事が多い。

4.
　Ａ（中止）のみ選択可能で異常終了して、データキャッシュが
　オフになっているか、「２」と同じハングアップ状態になる。

Human68k がv3.02になると症状悪化の傾向があるようです。

こんなので、問題の糸口はあるかなあ。

じゃぎゅあ

もうボロボロです。

細かなバグ取りをしただけで、動作の根幹にかかわるような変更はしてないはずなのに、これはどうしたことだ。

気になって仕事も手につきません。家に帰らなければプログラムリストもないので、頭の中で変更箇所を思い出してあれこれ考えてみますが、全然思い当たるふしがありません。夕方、仕事を終えてNIFTY-Serveにアクセスすると、有力な情報が寄せられていました。

187/187　NBH02724　じゃぎゅあ　040SYSpatch.sys 新版
（３）　94/01/26　17：39

Human68k（v3.01）＋040SYSpatch.sys（v2.2a）が異常になる
理由のよーな部分が分かりました。

040SYSpatch.sys（v2.2a）の1525行前後のHuman68k（v3.01）へのパッチがv2.1f から変更になっているところ。

Human68k（v3.02）で異常になるのも同じ処理してるところで、この場合はIOCS－AC が関係してくる。

もしかしたらIOCS－AC が？それともXC68040の○○？

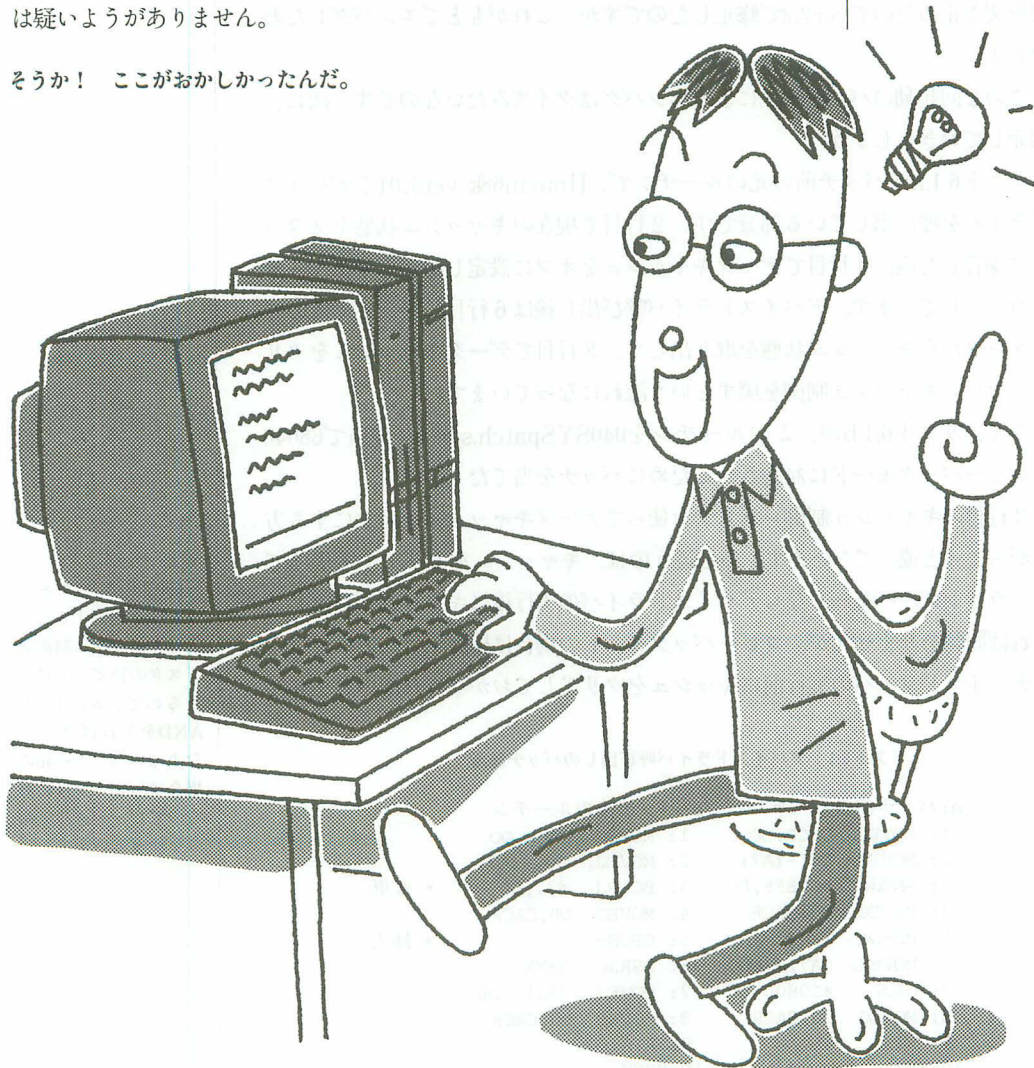
ということで、じゃんくしょっぷに新版いれときます。

Human68k（v3.02）の時のパッチは強引なものなので、もう少しIOCS-AC周りを検討しなければなりません。でもパッチすることで正常にはな

っている。

じゃぎゅあ

修正版までアップされています。とりあえず、これをダウンロードしてver 2.2aのどこが問題なのか、ソースの違いを調べてみました。すると、ver2.1aにあった040SYSpatch.sysが2回起動されてしまうというバグを直すために修正した部分が元に戻っています。たいした修正ではないからと疑っていませんでしたが、これを戻すとまともになるのですから、原因はこの修正にあるのは疑いようがありません。

そうか！　ここがおかしかったんだ。

バグの謎解き

ちなみに、この2回起動のバグとは、device文で1つしか指定していないはずなのに、なぜか040SYSpatch.sysが2回起動されてしまうというものです。2回目はコピー領域の確保に失敗するのでエラーになるのです。もっとも、1回目で必要な処理が完了しているのでエラーになっても実害はなく、無視すればすんでいました。

さて、この2回起動の原因ですが、これまた全然わからず、ver2.1aではあきらめてKnownバグとしてそのまま公開してしまいました。その後、この原因を突き止めたのでver2.2で修正したのですが、これがもとでエンバグしたわけです。

この2回起動のバグと修正によるエンバグはクイズみたいなのです。次に、紹介しておきましょう。

リスト6.1.aがパッチ前の元のルーチンで、Human68k ver3.01でデバイスドライバを呼び出している部分です。2行目で現在のキャッシュ状態をスタックに保管した後、4行目でデータキャッシュをオフに設定してからサブルーチンコールしています。デバイスドライバ呼び出し後は6行目に戻ってくるので、スタックからキャッシュ状態を取り出して、8行目でデータキャッシュをクリアしながらキャッシュ制御を戻すという流れになっています。

さて、リスト6.1.bが、このルーチンを040SYSpatch.sys ver2.1aで68040のコピーバックモードに対応させるためにパッチを当てたものです。

3行目のキャッシュ制御レジスタを使ってデータキャッシュをオフにする方法がちょっと違って*1います。あと違うのは、キャッシュをクリアする場所です。ライトスルーモードならデバイスドライバの実行後にキャッシュをクリアすれば問題ありませんが、コピーバックモードの場合は検討項目にあったようにデバイスドライバの実行前にキッシュをクリアしておかなければいけません。

リスト6.1　デバイスドライバ呼び出しのパッチ箇所

```
a:パッチ前のルーチン          b:パッチ後のルーチン
 1: MOVEC   CACR,D0           1: MOVEC   CACR,D0
 2: MOVE.L  D0,-(A7)          2: MOVE.L  D0,-(A7)
 3: AND.W   #$FEFF,D0         3: BCLR.L  #31,D0        ←変更
 4: MOVEC   D0,CACR           4: MOVEC   D0,CACR
 5: BSR.W   XXXX              5: CPUSH                 ←挿入
 6: MOVE.L  (A7)+,D0          6: BSR.W   XXXX
 7: OR.W    #$0800,D0         7: MOVE.L  (A7)+,D0
 8: MOVEC   D0,CACR           8: MOVEC   D0,CACR
 9: RTS                       9: RTS
10:XXXX:....                 10:XXXX:....
```

＊1
68030のデータキャッシュを制御するビットはキャッシュ制御レジスタの12ビット目にあるので、$FEFFとANDをとればオフになりますが、68040の場合は31ビット目にあるのでビットクリア命令を使っています。

そのために、5行目の部分に無理やりcpushを挿入しているのです。このため、その次のサブルーチン呼び出しが1つ後ろにずれています。

そして、これが040SYSpatch.sysの2回起動というバグを生んだのでした。ちょっと理由を考えてみてください。

さて、わかったでしょうか？

ポイントはbsrの移動です。

Human68k内部のリスト6.1.aのルーチンをパッチして、リスト6.1.bのように書き換えているわけですが、この処理をするのは040SYSpatch.sysです。040SYSpatch.sys自身、デバイスドライバであるため、最初はリスト6.1.aのルーチンで呼び出されます。このときはbsrは5行目にありますから、戻り番地は6行目としてスタックに積まれています。ところが、040SYSpatch.sysでパッチするので、デバイスドライバから戻ってきたときはリスト6.1.bのルーチンに書き変わっており、戻ってきたつもりの6行目で再度bsrが実行され、2回目の040SYSpatch.sysの呼び出しが起こるというわけです。

自分でミスったとはいえ、この2回呼び出しのバグはなかなか見つけられず、苦労しました。わかってしまえば簡単ですが、こういうトリッキーなことをするプログラムでは、バグのほうも一筋縄ではいきません。

COLUMN

Human68k ver3.02の場合

本文で説明したようにver3.01の場合は、デバイスドライバ呼び出しの前後でキャッシュ制御コードをじかに使っていましたが、ver3.02は「move from SR」命令の対処の部分にしかキャッシュ制御コードは使われていません。

これは、次のようにIOCS-＄ACを使うように書き改められているからです。

- IOCS-＄AC（D1＝1）をコールして現在のキャッシュ状態をスタックへ保存
- IOCS-＄AC（D1＝4）をコールしてデータキャッシュをオフに設定
- デバイスドライバ呼び出し
- IOCS-＄AC（D1＝3）をコールしてキャッシュをクリア
- IOCS-＄AC（D1＝4）をコールしてスタックに保存しておいたキャッシュ状態に戻す

IOCS-＄AC（D1＝4）のキャッシュ設定コールは、キャッシュプッシュもするように修正されている*1ので、これで問題ありません。

＊1　もちろん、これを修正しているのは040SYSpatch.sysです。

このbsrの位置がまずかったことがわかったので、ver2.2ではbsrの位置がずれないように気をつけました。といっても、cpush命令をどこかに挿入しなければなりません。そこで、リスト6.2のようにデバイスドライバから戻ってきたときの処理を詰めて、bsrの飛び先を10行目から9行目に変更し、ここでcpushをするようにしたのです。

これで確かに2回起動はなくなりました。めでたしめでたし、と思っていたのですが、これがコピーバックモードでの不調を引き起こすことになりました。

こちらもちょっと考えてみてください。

わかりづらいので、1つずつ説明していきましょう。

最初に、4行目でデータキャッシュがオフになります。このとき、ダーティキャッシュは宙ぶらりんの状態になってしまいます。前は、ここですぐcpushをして書き戻しをしていたのですが、その前にbsrが入りました。bsrが実行されるとき、6行目に戻ってこられるよう、戻りアドレスがスタックに積まれるのですが、この状態ではデータキャッシュはオフになっているので、キャッシュには手をつけず、スタックポインタで示されるメモリに直接戻りアドレスが書き込まれます。

その後、9行目でcpushが実行されるのですが、このとき、たまたまダーティキャッシュとして今書き込んだスタックポインタのアドレスと同じアドレスを指しているものがあると、こちらを書き戻そうとして6行目への戻りアドレスを上書きしてしまうのです。

前にcache.xの不具合*1を説明したときのプロセスは、これとは逆です。あちらは書き戻されなかったために、古いデータをアクセスしておかしくなっていましたが、こちらは新しいデータを書いた後に古いデータを書き戻してしまったためにおかしくなったわけです。

＊1
187ページ「cache.xでエラー」参照。

リスト6.2　デバイスドライバ呼び出しのパッチ箇所

```
ver2.2で修正した後のルーチン
 1:          MOVEC     CACR,D0
 2:          MOVE.L    D0,-(A7)
 3:          CLR.L     #31,D0
 4:          MOVEC     D0,CACR
 5:          BSR.W     XXXX
 6:          MOVE.L    (A7)+,D0
 7:          MOVEC     D0,CACR
 8:          RTS
 9:  XXXX:   CPUSH                         ←ここに移動
10:          ....
```

結局、データキャッシュをオフにしてからキャッシュプッシュするまでの間は、メモリに対するデータアクセスをしてはいけないということなのですが、サブルーチン呼び出しをうっかり入れてしまったというわけです。

う〜ん、コピーバックは奥が深い。

わかったつもりになっていたコピーバックでしたが、ver2.1のときは施したパッチがたまたまこの問題に引っかからないようになっていただけで、本質的な問題を理解したうえでパッチしていたわけではありませんでした。

原因がわかったところで、NIFTY-Serveにアクセスすると、すでにその対処も行われた新版がアップロードされていました。

```
188/193　　NBH02724　じゃぎゅあ　　　　040SYSpatch 新版
（3）　　94/01/26　21：33　　　　　　　　コメント数：　1

新版です……。
夕方にダウンロードした方、すいません。

キャッシュ関係で異常になる理由が判明したため、そこを修正
してあります。

Human68k の根幹に関わるよーな部分だったので、今まで不安定
だったものが動くようになっているかもしれません。

じゃぎゅあ
```

素早い！

さっそくver2.2cをダウンロードします。ソースを見てみると、ほぼ完璧でした。ただ、ver2.1と同じ処理に戻したため、040SYSpatch.sysの２回起動の問題が復活していました。

２回起動のバグの原因を報告していなかったので、ついでに直してもらおう[*1]と報告をすると、じゃぎゃあ氏はこれにも素早く対応してくれました。

しかし、結局、ほかにもいろいろ問題が出てしまい、最終的に２回起動を直すかわりに、２回目の起動を無視する方向で対応してもらいました。

たいした修正じゃないはずだったのですが、結果的にver2.1の時点で隠れて

＊1
以後、いろいろ忙しくなったこともあって、040SYSpatch.sysの対応は、じゃぎゅあ氏に全面的に頼っています。

いた問題が明るみに出て、コピーバックの問題点をいろいろ考えさせられました。

修正版を作ってくれたじゃぎゅあ氏をはじめ、バグレポートをしてくれた多くの人の協力があったから、なんとか、ここまでこぎつけることができたわけで、1人だったら、とうの昔に投げ出していたでしょう。

3度目のハードウェア

第一次、第二次配布を通じて、さほど大きな問題*1もなく、比較的順調にきましたが、そうなってくると、細かい問題が目についてきます。

報告が多かったのが、「VRAMにゴミが出る」というものでした。最初はクロックアップしている人からの報告だったので、クロックアップの弊害*2と片付けていたのですが、どうもクロックアップしていない25MHzのままのマシンでも発生することがあるようです。

ただ、これもマシンの個体差によるもののようで、全然出ないという人もいれば、結構目立つ人もいました。何かの拍子に出る人もいましたが、いまいち因果関係がつかめていませんでした。

致命的ではないので後回しになっていたのですが、余裕も出てきたので、そろそろ対応を考えようと思っていたところ、おもしろい報告が届きました。

＊1
「68030ぴーんち」が、今までで一番大きな問題でした。

＊2
68030でもクロックアップするとVRAMにゴミが出ます。

213/243　GBD02245　－ O・U －　040でのグラフィックのゴミ
(3)　94/01/29　02：53　コメント数：　1

GRADを使ってメモリ頭から100KByteずつRAMDISKを確保していって，MEMFREEでの使用可能領域表示が8538688バイトになったところで，HAPICによるグラフィック表示でゴミが出始め，7719488でゴミがでなくなります。その後，また4330064でゴミが出始め，3510864でゴミがでなくなります。ちなみにメモリはI/O DATA製のやつで，使用グラフィックはサイバリオンのロード時に出る絵です。また，HAPICだからゴミが出るわけではなく，HAPICは表示が速くゴミが出るとわかりやすいということによります。絵も別に他の絵でもゴミは出るのですが，これがサイズもでかく，ゴミが出たのか出ないのかわかりやすいからです。これがHAPICではなくAPICGだとゴミの出方が変わります。難しいのですが，HAPICだと上から表示して行くごとに下の方にゴミがたまって行って，最終的にそのゴミが結局(表示させた)絵で書き換えられてしまうといった

感じで，APICGだとゴミは減るのですが，絵の表示がおかしいというか
なんというか。。。APICGは内部で展開して転送しているみたいな感じ
だから、その辺の違いによるものかも知れません。
なお，クロックを上げるとこの限りではないようです。かなりゴミが出
るようです。ちなみにSW１はショートしてません。

これが一体何を意味するのか，私には皆目見当もつきません。。。

でもって，BEEPsさんに以下を質問されたので。

1）キャッシュオンの話しですよね？　キャッシュオフでも出ますか？
どっちでも出ました。それにしても，キャッシュオフだと遅い。。

2）コピーバックモードですか、ライトスルーモードですか、どちらで
も出ますか？
ライトスルーです。コピーバックにはどうするとなるのかよくわか
らない。

3）クロック25MHzでも出ますか？
25MHzで出ます。なお，キャッシュオフでwait 15でやってみ
たのですが，ゴミはやっぱり出ます。

4）クロックはICクリップで取り出していますか？
私はクロックの２段階切り替えなることをしているため，クロック
は切り替え回路の元々オシレータが刺さっていたところのピンのと
ころにはんだ付けしてます（意味不明？）。74F803のところにはんだ
付けしようかとも思ったのですが，これ以上基板上にいじるのは気
が引けたので。。。

ちなみにほとんど何もデバドラ関係組み込まなくってもゴミは出ました。
このときはdcache２aでメモリ埋めましたが。でも，memfreeの表示は少
しだけずれてたくさい。誤差？

もう一つ。030のモードではゴミは出ません。SW１のショート取っち
ゃったので，クロック上げたらどうなるかはわかりません。が，SW１シ
ョートしていた当時はクロック上げてもゴミは出てませんでした。

#　こんなのうちの機械だけかなぁ。

でわ。

おゆ

どうもVRAMに描画するプログラムの実行アドレスに依存するようです。しかし、同じような状況にしても、家のマシンでは、やはり発生しません。

困ったなあ。再現するマシンを送ってもらうしかないか?

おゆ氏に本体貸し出しをお願いするかもしれない旨のメールを出します。

クロックアップとマウスのゴミ再発

おゆ氏の承諾は取り付けましたが、まだ、家のマシンで試すことが残っていました。クロックアップです。クロックアップしたマシンでは軒並みVRAMにゴミが入るということなので、まずは家のマシンをクロックアップすることにしました。

まだ保証期間内の[*1]マシンのマザーボードに手をかけるのは気が進まないのですが、いずれクロックアップするつもりだったので、それがちょっと早まっただけだと自分に言い聞かせて、マザーボードのオシレータを引っこ抜き[*2]ます。

マザーボードのオシレータ部分をICソケットに取り替えて、オシレータを取り替えられるようにしたうえで66MHzのオシレータを挿して実行してみました。これならクロックは33MHzになります。オシレータの取り替え以外、特に何もしませんでしたが、問題なく[*3] 68040が起動しました。

しかし、肝心のゴミが出ません。

もっとクロックアップしないとダメなのかなあ。

そう思っていたのですが、プログラムの実行アドレスに依存することを思い出し、キャッシュオフ[*4]にしてみました。

すると、マウスカーソルを表示させただけで、チラホラとゴミが出てきます。前にキャッシュオンにするとマウスカーソル表示でゴミが出る問題がありましたが、今度は逆にキャッシュオフにするとゴミが出てきてしまいました。

おお、ゴミじゃゴミじゃ。

と、しばらく喜んでマウスをグリグリ動かして遊んでしまいました。

＊1
これだけいろいろいじくりまわしておいていまさら保証期間を気にするのも変ですが、いちおう、いつでもまっさらな状態に戻せるようにしながら作業していたのです。

＊2
もちろん、半田ゴテで半田を溶かしながら、です。

＊3
68030でクロックアップするためには、いろいろ手をかけなければいけないようですが、68040モードは68030の動作よりもウェイトが入っている分、クロックアップしても安定した動作をするのです。

＊4
キャッシュオンだとプログラム読み込みのためのメモリアクセスが減るので、実行アドレスへの依存度が下がると考えられます。

VRAMゴミ問題

さて、ゴミが出たのはいいのですが、いまいちハデに出てくれません。画面の下のほうの数ラインにポロポロ出るだけです。

実行アドレスに依存しているはずですから、マウス描画ルーチンのあるアドレスを変えてみることにしました。

このルーチンは、前のマウスのゴミ問題で相当苦労して調べていたので、取り扱いは慣れたものです。マウス描画ルーチンを、フリーエリアの$300000、$310000、……、$3F0000の各々のアドレスにコピーし、本来のルーチンのほうにはJMP命令を埋め込んで、各々のアドレスでマウス描画ルーチンを実行させてみました。

すると、$370000番地にマウス描画ルーチンがあると、盛大にゴミが出ることがわかりました。また、$380000番地から実行させた場合は全然ゴミが出ません。この2つの実行アドレスの違いを見ると、アドレスのA19～A16のビットが7か8かの違いです。

テキストVRAMは$E00000～$E7FFFFのアドレスを占めており、このうち、マウスでゴミが出ているプレーンは$E60000～$E7FFFFのプレーンであるということとあわせて考えてみると、マウス表示のためにVRAMへの書き込みをした後、次のプログラムの読み込みをしようとしてアドレスが$370000に変わろうとしている部分で、$E60000～$E7FFFFへのゴミの書き込みが行われてしまうようです。クロックアップして、アドレス変化のタイミングが速くなったからか？　と思って、念のため25MHzに戻して試してみたところ、

ゲッ、25MHzでもゴミが出るじゃないか！

クロックアップ改造でオシレータをICソケットにしたのがまずかったかなと思ったのですが、テストプログラムを作り、みんなにキャッシュオフ状態で$370000番地から実行してもらったところ、やはり、ゴミが発生することがわかりました。それも、今までゴミに出合ったことがないという人のマシンでもゴミが確認されました。

「68030ぴーんち」のときのように、マザーボード上のVRAMのRAS (Raw Address Strobe) 信号にコンデンサをかますとゴミが書き込まれにくくなることもわかりましたが、すべての人にこの対処をやってもらうのは大変です。

それに「68030ぴーんち」と違い、こちらは68040モード側の問題なので、タイミングを変換回路で細工する[*1]ことができそうです。

＊1
68030モードでは、ほとんどの信号が直結されていて変換回路が間に入っていないので、タイミングを細工できないのです。

DLEモードの改造

原因の見当がついたところで、対処方法ですが、アクセスが完了してからも68030のようにアドレス信号を保持してやるのは容易ではありません。

68040はほとんど絶え間なくメモリアクセスをしようとしますので、アクセスが終われば、すぐに次のアドレスを出力しようとするからです。

しかし逆に、図6.1.aのようにX68030本体側にはAS信号をネゲートしてアクセスが終わったように見せながら、68040に対してはまだアクセスが完了していないように見せかければ、その間アドレス信号が変わることはありません。

もちろん、AS信号をネゲートすれば、データバス上のデータが無効になってしまうので、その前にデータだけは取り込んでおかなければなりません。そこで登場するのがDLE（Data Latch Enable）モードと呼ばれる特殊なアクセスモード*1です。

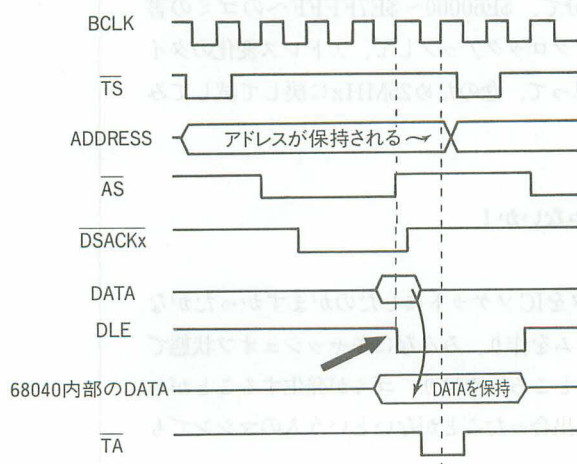
68040は、データバスの内部にデータを一時的に保持するラッチを持っており、アクセスのサイクルに関係なく、DLE信号をLowレベルにすると、そのときにデータバスのデータをこのラッチに保持してくれるのです。図6.1.aの矢印で示したタイミングでDLE信号をLowにしておけば、データは68040内部に保持されるので、AS信号をネゲートした後にTA信号をアサートしてもちゃんとデータを受け取ることができます。

*1
もともと、DLEモードは同期アクセスと相性の悪いデバイスを接続するためにあるので、最初からこれを活用すべきだったかもしれません。

図6.1　DLEモードのタイミング

DLEモードを使うようにしてアドレス変化のタミングをずらすようにしたところ、バッチリVRAMのゴミが出なくなりました。

しかし喜んでもいられません。この対処は、もちろんソフトウェアではすみませんから、3度目のGAL交換です。さらに悪いことに040turboの本来の回路ではDLEモードを使うようにはなっていませんでしたから、基板改造が必要になりました。

DLEモードを指定するには、68040のリセット中にMDIS (Mmu DISable) 信号をアサートしてやらなければなりません。この信号は、68030のMMUDIS (MMU DISable) 信号につないでいるだけでしたから、そのままではリセット中はネゲートされたままでDLEモードになりません。

最初は単純に考えて、MDIS信号を68040のRSTI (ReSeT In) 信号と接続したのですが、動かなくなってしまいました。DLEモードが変なのかと思っていたのですが、実はこれはX68030側の回路に問題がありました。どうもX68030側でMMUDIS信号やCDIS信号をまとめて1つの抵抗でプルアップしているらしいのです。このため、68040側でMDIS信号をリセット中にアサートすると、X68030側を回って68040側のCDIS信号までアサートされてしまいます。CDIS信号をリセット中にアサート*1すると、こちらはアドレスバスとデータバスを時分割で使う多重化バスモードになってしまいます。

X68030の回路*2を恨んでもしかたがありません。68040のMDIS信号を、X68030のMMUDIS信号につないだままではダメですから、写真6.1.aのように、040turbo基板上のパターンを切断したうえで、写真6.1.bのように、RSTI信号と接続します。もう1本の接続線はDLE信号をGALに接続している線です。

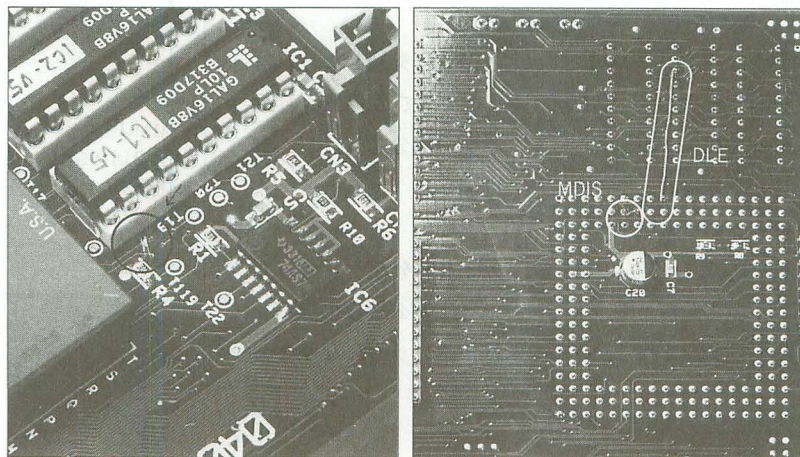



写真6.1　改造箇所　a.MDIS信号のパターンカット(左)　b.改造線を張ったところ(右)

＊1
ほかにIPL 0－2信号などをリセット中にアサートすると、ラージバッファモードになります。

＊2
68030のユーザーズマニュアルに、68030を68020のシステムにつなぐための回路例として載っていたので、これに倣ったのでしょう。

結局、

・MDIS信号とMMUDIS信号のパターンをカットする

・MDIS信号をRSTI信号に接続する

・DLE信号を、新たにGALで作り出した信号ピンに接続する

この3点の改造を施さなければならなくなりました。

タイミングの改良

VRAMのゴミが出なくなったのはいいのですが、X68030側にアクセスが完了したように見せてから、さらに68040側は1クロック待つわけですから、またまたウェイトが増えてしまいました。

これで合計3ウェイト。いくらなんでも多すぎます。今までは性能は二の次、とにかく安定して動けばよいという方針できましたが、VRAMのゴミと性能を秤にかけると、ちょっと厳しいところです。

しかし、考えてみるとメインメモリアクセスでは別に問題は出ていませんでしたので、VRAMアクセスのときだけ1クロック待つだけでもすませられそうです。

実際にはVRAMアクセスだけというのは面倒なので、*1DSACKx信号で16ビットサイズのデータバスだった場合*2には1クロック待つということにしてみました。

試してみたところ、これでもVRAMのゴミが出ませんし、メモリアクセスもほぼ今までどおりです。

よしよし、これで完璧だな。

そう思って、これで対処しようかと思いましたが、GALを交換しなければいけないことには変わりがありません。かつて同じようなミスで2回*3もGALを交換していますから、今回のGAL交換はあわてずに他の部分で改良ができないかどうかも含めて、じっくり検討してみることにします。

まずは、今ある、2クロックのウェイトです。

ためしに、TS信号からAS信号を作り出すときのウェイトをなくしてみたところ、あっさり動いてしまいました。どうもこのウェイトが必要なのはバラック基板の問題*4だったようです。これで、ウェイトは1クロックになりました。

これなら、ノーウェイトでいけるかもしれない！

＊1
変換回路はアドレスをチェックしていないので、VRAMアクセスかどうか判断できないのです。

＊2
DSACK0がHigh、DSACK1がLowという組み合わせです。

＊3
MMUとコピーバックの2回のGAL交換がありました。

＊4
グチャグチャの配線だったので、アドレスが安定するのが遅かったのでしょう。

今度は、本命のDSACKx信号から、TA信号を作っている部分のウェイトを削ってみました。

しかし、さすがにこちらはエラーです。ただ、まったくまにあわないようにも見えません。ほんのわずかのタイミングで何か失敗しているようです。いろいろ調べていくと、DSACKx信号からデータバスのサイズを決定しているところに問題があることがわかりました。

DSACKx信号はDSACK0とDSACK1の２本の信号からできていて、どちらかがLowレベルになって応答するとともに、この２本の信号の組み合わせでデータバスのサイズを示すようになっています。ここで、変換回路は、DSACK0信号とDSACK1信号をクロックの立ち下がりエッジでフリップフロップに取り込み、データバスのサイズを決めていたのですが、どうもDSACK0とDSACK1が同時にLowレベルにならない場合があることが見つかりました。図6.2のようにDSACK1がLowになった後でDSACK0がLowになるというように微妙にずれる*1と、本来はロングワードアクセスであるはずが、ワードアクセスと思って変換回路は誤って動いてしまうのです。

２つの信号がずれてもいいように、変換回路のGALのプログラムを修正してみたところ、DSACKx信号からTA信号を作る部分の１クロックのウェイトをなくしても、問題なくアクセスできるようになりました。

*1
68030のタイミングチャートではずれを許しているので、これでも問題ありません。

これなら、68030とほぼ同じメモリアクセス速度だ。

図6.2　アクノリッジ信号のずれ

ためしにメモリアクセスプログラムを作って実行してみたのが、表6.1です。68030の命令キャッシュ、データキャッシュの両方がオフであるときのDRAMアクセス時間を１として、その他の場合の相対的なメモリアクセス時間を表しています。旧GALの68040のメモリアクセスは、２クロックのウェイトが足を引っ張って、68030よりも時間がかかっていましたが、新GALでは改善され、68030よりも短くなっています。といっても、実際のメモリアクセスに要するクロック数は同じはずです。何が起こっているのか[*1]と思って、実際のアクセスの様子をロジックアナライザで観察してみたところ、68030が１命令１命令の実行の間に内部処理のための空き時間が入る[*2]のに対し、68040ではメモリアクセスと内部処理が並行して実行されたおかげで、メモリアクセスが連続的に実行されていました。このため、相対的にメモリアクセスの時間が短縮されたように見えるのです。これは、命令キャッシュをオンにするとさらに顕著になり、DRAMアクセスに関しては68040のメモリアクセス時間は68030の半分以下になったように見えます。

＊１
といっても、単純に命令実行時間の違いではありません。この表の値は、メモリ転送命令を1000000回実行するのに要した時間から、レジスタ転送命令を1000000回実行するのに要した時間を引いて求めているからです。

＊２
ウェイトが入るVRAMアクセスの場合は、内部処理がウェイト中に実行されてしまうので、差が縮まります。

表6.1　メモリアクセス時間の比較

	キャッシュOFF		命令キャッシュON	
	DRAM	VRAM	DRAM	VRAM
68030	1	1.52	0.63	1.31
68040＋旧GAL	1.11	1.88	0.41	1.32
68040＋新GAL	0.83	1.49	0.32	1.12

68030キャッシュオフ時のDRAMアクセス時間を１とする。

クロックアップへの対応

これで完璧かと思ったのですが、やはり問題がありました。クロックアップに耐えられなくなっていたのです。もともとの040turboのGALは２クロックのウェイトがあったので、68030よりもクロックアップに対するマージンが大きく、余裕でクロックアップできていたのですが、ウェイトを削った分、マージンが減ってしまったのです。

本来、X68030は25MHzのクロックで動作しているのですから、このクロックでベストの性能が出れば、後のクロックアップでの性能はどうでもいいといえなくもないのですが、すでにこのときは家のマシンもクロックアップの改造をしていたので、25MHzだけにこだわってはいられません。

せっかく削ったウェイトですが、設定によってウェイトを入れられるようにしました。25MHzで満足な人はウェイト無しの状態にし、クロックアップしたい人はウェイトを入れる設定をすればいいのです。GALの空き端子で、こ

の設定をするようにしました。

1. IC2の9番ピンをLowにする
DSACKx信号からTA信号を作る間に1クロックウェイト
2. IC1の5番ピンをLowにする
TS信号からAS信号を作る間に1クロックウェイト

両方の設定をすれば、ほぼ、前のGALと同じタイミングになりますが、1.の設定だけで36MHzでも十分動き*1ました。

＊1
68040モードは問題なく動きますが、68030モードは起動すらしません。

実験

これらの対処で、ほぼ納得のいくものとなりましたが、やはり、まだ油断はできません。なんといっても、わが家のマシンで問題が出なくても、他の人のところで問題が出ないとはいえません。

今回は基板改造という作業も待っていますから、大事をとって何人かに試してもらうことにしました。

NIFTY-Serveで参加者を募ったところ、じゃぎゅあ氏とおゆ氏が参加してくれました。また、クロックアップでいろいろ試してもらっていたY.氏、および名古屋近郊ということでSUPRA氏には直接依頼して「人柱」*2になってもらいました。

＊2
実験参加者は「人柱」と呼ばれていました。

その結果、「VRAMゴミ問題」は解消され、クロックアップした場合でも1ウェイト入れれば大丈夫ということが確認されました。基板改造について、自分でやるのは不安だという人の分は改造を請け負い、自分で改造できる人には、対処GALを焼いたものを送って、交換してもらうことになりました。

C O L U M N

GAL焼き

もちろん「焼く」というのは、書き込みをするのに高電圧をかけることから転じた、いわゆる業界用語です。実際、ROMなどはデータが多くて書き込み時間がかかるので、書き込み後すぐに触ると結構熱くなっているのがわかります。

040turboでは、すべてのGALのソースと、GALライタにかけるためのJEDEC形式というファイルを公開していますので、GALライタさえあれば自分でGALを焼くことができます。しかし、いくらマニアックなユーザーが多いといっても、さすがにGALライタを持っている人はそう多くはいません（それでも、参加者のうちに何人かいたのが凄いところです）。

BEEPs氏、GALを焼くの図

第7章

040turboとX68の可能性

040turboは大丈夫か?

040turboの製作過程を駆け足でお話ししてきましたが、いかがだったでしょうか。この本*1を書いている今も、実は「VRAMゴミ問題」対処作業のまっ最中です。今回は基板の改造作業を各自でやってもらい、それが完了した人から順に変更になったGALを発送する*2ということにしたので、まだすべての差し替えが終わっていません。

今度こそ完璧だ。

と思ってやってきたのに、やっぱり問題*3が発生していますから、また、別の問題が出てくるかもしれません。個人の趣味で作ったものですから、基本的に「動けばいい」ということで設計していますし、第一、X68030側のハードウェア、ソフトウェアの情報が不足していますから、推測で補っている部分が多数あります。そういう意味で「絶対大丈夫」と保証することは永遠にできないでしょう。

しかし、フリーソフトウェアが、公開後いろんな人に使われて徐々に安定したものになっていくように、040turboも第一次、第二次の参加者によって使われることで多くの問題を発見し、克服してきました。ハードウェアの3回のバージョンアップだけでなく、ソフトウェアのほうもバージョンアップを頻繁に行いました。その甲斐あって、今では68040モードが当たり前で、わが家のX68030が本来の68030モードで使われることはゲームのとき以外はほとんどありません。それくらい安定して*4動いています。

また、Human68kについてしか触れてきませんでしたが、040turbo対応のNetBSD*5など、まだまだ040turboのソフトウェアについてはおもしろい話がたくさんあります。これらは、フリーソフトウェアの作者の方々にお願いして寄稿してもらった「番外奮闘編」にまとめましたので、そちらに譲るとして、最後に、ハードウェア工作のおまけとして次の4点を紹介しておきましょう。

・倍クロック回路
・クロックアップ改造
・ハイレゾ改造
・040turbo on X68000

＊1
1994年4月現在、予定は大幅に遅れています。

＊2
一気に全員分を差し替えられるほどGALの予備がないのでしかたがありません。

＊3
正規の製品だったらひんしゅくものです。

＊4
付録に040turboでのアプリケーションの動作状況をまとめておきました。

＊5
フリーのUNIXの1つです。X68030版は、沖氏を中心として移植が進められています。

倍クロック回路

現状の040turboの最大のウィークポイントは、68040のPCLK信号に必要な50MHzクロックを、X68030のマザーボードの50MHzのオシレータからICクリップを使って直接取り出している点です。もっとも、実際の動作は当初心配していたほど不安定ではなく、25MHzの通常のX68030なら問題ないようでした。さすがにクロックアップ改造しているマシンだと動作が不安定で、これは、ICクリップをやめて半田付け*1してしまえば解決するという見通しから、なんとか今の方式で動かすことを考えています。

ただ、こういう形での電気信号の伝送は、もともと胸を張れたものではありません。このため、別の手段でPCLK信号を作り出す方法を実験することにしました。これが、倍クロック回路です。

040turboのバラック基板で最初にやったように、安易な逓倍回路では安定した倍クロックは得られませんので、今度は真面目に倍クロック専用のLSIを使ってみました。モトローラのMC88915*2チップがそれです。入力クロックの倍のクロックを生成するだけでなく、設定によって２分の１のクロックや４倍のクロックも生成することができるという優れものです。これを使用すれば、X68030の68030ソケットからくる25MHzのクロックをもとに、68040のPCLK

＊１
このことから判断すると、不安定動作の原因は、クロックアップしたからというよりも、オシレータを取り替えられるようにICソケットを使わざるおえなかったからかもしれません。

＊２
クロックシンクロナイザという名前がついています。

C O L U M N

040SYSpatch.x

040turbo対応プログラムの核となる040SYSpatch.sysはじゃぎゅあ氏の手によって大幅に改良が加えられ、ver2.5からは〝040SYSpatch.x〟という〝.x〟の拡張子の実行ファイルになっています。といっても、パッチプログラムとしては従来どおりデバイスドライバとして登録しなければなりません。コマンドラインから実行した場合はバージョン表示をするようになっています。

改良点は多々ありますが、特に便利になったのはROMのパッチ後にリセットなしに１発で起動するようになったことです。MMUのアドレス変換機能を使って、ROMコピー領域を本来ROMが存在するアドレスにマッピングするようにしたため、ROMコピー領域でリセットし直す必要がなくなったのです。

ほかにも、懸案だったデバイスドライバ呼び出し時のキャッシュクリアをパスさせることでディスクアクセス時のキャッシュクリアによるパフォーマンス低下が大幅に改善されています。

信号に必要な50MHzのクロックを040turboの基板上で生成することができるというわけです。実は、このやり方は「68040デザイナーズハンドブック」にも使用例が紹介されています。

「じゃあ、なんで最初からこれを使わなかったんだ？」という疑問を抱かれるかもしれませんが、バラック基板を作っているときはMC88915を入手できず、触ったこともなかったのです。使ったことのないチップは怖くて使えません。

ところが、040turboの基板配布を始めたことでいろいろな情報が集まってくるようになりました。そのなかで、50MHzのクロックを得る改良案として桑野氏[*1]が、このチップを使って実験してくれました。また、PA５氏[*2]は、同じく倍クロック生成用のサイプレスのCY7B991-7JCというチップで実験してくれました。

しかし、両方ともうまくいきませんでした。桑野氏からはMC88915と実験回路を送ってもらって私自身もあれこれ試してみましたが、やはりダメでした。50MHzともなるとわが家の80MHzのロジックアナライザではまともに観測できませんので、デバッグのしようがありません。

100MHzくらいのオシロスコープがほしいなあ。

そんなことを思いながら、しばらく放っておきました。

クロックアップと倍クロック回路の関係

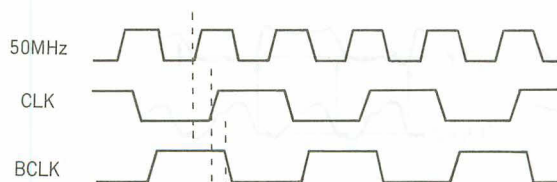
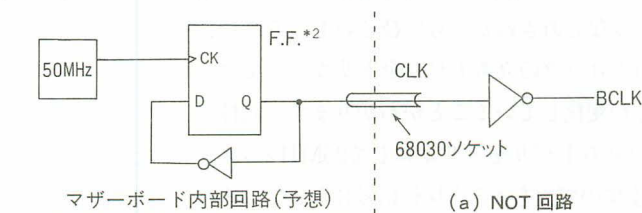
ところが、あるとき、ひょんなことから、これが日の目を見ました。なんとクロックアップしていると、この倍クロック回路で040turboが動作するようになったのです。

もともとは「VRAMゴミ問題」の対処のためにわが家のX68030をクロックアップしたのですが、この問題が片付き、やれやれと思っていたとき、ついでに倍クロック回路を試してみようと思い、つないでみるとあっさり動くではありませんか。このときは、50MHzのオシレータのかわりに72MHzのオシレータをつないで36MHzにして動かしていたのです。ためしに50MHzのオシレータに戻してみると、やはり動かなくなりました。倍クロック回路は確かに動いていたのです。動かない原因は040turboのほうにあったわけです。「36MHzで動いて25MHzで動かない」というのはなんとも奇妙な話ですが、ここで、はた、と気がつきました。それは、040turboで使っている反転クロックの問題です。

＊1
『Inside X68000』、『Outside X68000』、『X68030 Inside/Out』の著者として有名ですね。

＊2
ジャンク品の68040を安価で提供してくれた人です。

68040の信号はBCLK信号の立ち上がりが基準になっているのに対し、68030の信号は主にCLK信号の立ち下がりで動作します。このため、040turboではマザーボードからの68030のCLK信号を反転させ、これをBCLK信号として68040に与えているのです。普通、反転信号を得るためには、図7.1.aのようにnot回路を使うのですが、それだと図7.1.bのようにnot回路の遅延分だけ反転した信号が遅れてしまいます。これを嫌って、040turboでは図7.2.aのような回路で元のクロックの反転信号を得ているのです。元のCLK信号自体がX68030のマザーボード上の50MHzクロックをトリガにしてフリップフロップ回路で分周して作り出されているので、同じように、この50MHzのクロックをトリガとしCLK信号をフリップフロップで取り込んでやります。こうすれば、CLK信号もBCLK信号も同じようにフリップフロップを１個通ってくることになる*1ので、図7.2.bのように元の50MHzのクロックに対し、ほぼ同じ遅れで反転した信号になります。

しかし、この反転クロックの生成の凝った作りが仇になりました。倍クロック回路で作られる50MHzのクロックは25MHzのクロックをもとに生成したものですから、むしろ遅れるのは50MHzのクロックのほうです。このため、正常な反転クロックを得ることができなくなるのです。

＊1
マザーボード上でCLK信号を作っているフリップフロップは74F803というICを使っています。一方、040turboでは74AS74というICを使っているので、実際には、多少の違いは出ます。

(a) NOT回路

(b) 波形

図7.1　not回路による反転信号

図7.2　040turboの反転信号

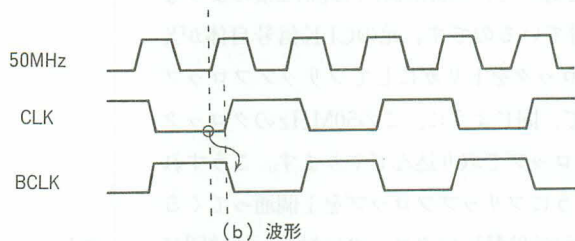
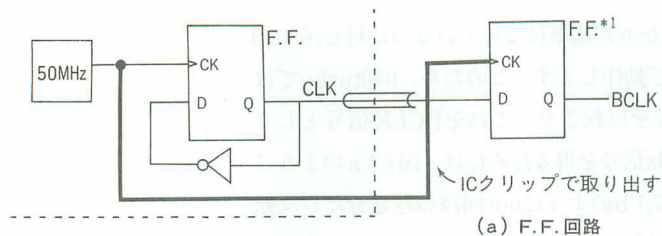
＊1
74AS74というフリップフロップを使っています。

そして、実際に倍クロック回路の波形をPA５氏に測定してもらった結果が図7.3です。これはCY7B991-7JCの波形ですが、MC88915の波形も同じようになっているのでしょう。

図7.3.aが、マザーボードから50MHzクロックを取り出した場合の波形です。50MHzの波形はよくこれで動いているなとあきれるくらいひどいものですが、それはおいておいて、この50MHzのクロックの立ち上がりをトリガにしてマザーボードの25MHzのクロック信号が変化していることがわかります。同様に、反転クロックも、この50MHzの立ち上がりをトリガにして25MHzのクロックの矢印の部分を取り込んでいますので、ちゃんと反転信号になっています。

（a）ICクリップ取り出しによる波形　（b）倍クロック回路による波形

図7.3　040turboクロック波形

ところが、倍クロック回路で作った50MHzでは、こうはいきません。図7.3.bを見るとわかるように、25MHzで生成した50MHzのクロックの立ち上がりは元の25MHzのクロックより若干遅れているのです。このため、反転クロックは元の25MHzクロックが変化した後の矢印の部分の信号を取り込んでしまい、反転信号にならずに元の信号と同じ波形になってしまっているのです。

倍クロック回路の試み

さて、原因がわかったところで対処方法ですが、PA５氏は、倍クロック回路で生成する50MHzのクロックを、図7.3.aと同じように元の25MHzのクロックよりも若干早めに出るように調整して成功させました。CY7B991-7JCは、生成するクロックのスキューを調整できるのです。図7.4がこのチップを使った回路図、写真7.1がこの回路を取り付けた040turboです。

一方、MC88915は倍クロックのスキュー調整ができませんが、かわりに反転クロックが出力されているので、040turbo本来の反転クロックの生成回路を殺して、MC88915が作った反転クロックを使うようにしてみました。図7.5が040turboのコントローラ部をMC88915対応に改造した回路図です。

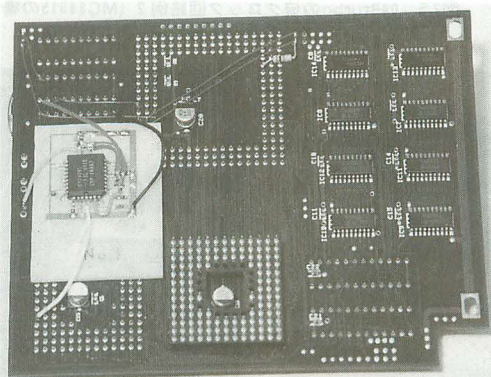
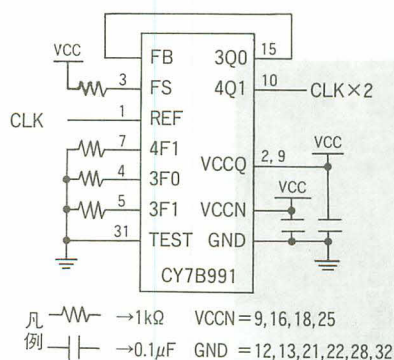



図7.4　040turboの倍クロック回路例１（CY7B991の場合）

写真7.1　倍クロック回路を取り付けた040turbo

これをもとに作ってみたのが、写真7.2の倍クロック回路です。無事、こちらも動かすことができました。

倍クロック回路を使えば、ICクリップを使ってマザーボードから50MHzクロックを取らなくても040turbo上で作り出すことができ、マザーボードの68030ソケットへの取り付けだけですむようになります。プリント基板のアートワークをやり直す機会*1があれば、この回路を取り入れたいと思っています。

＊1
当分ないとは思いますが。ちなみに、現状の040turboに取り付ける形の倍クロック回路をオプションボードとして検討中です。

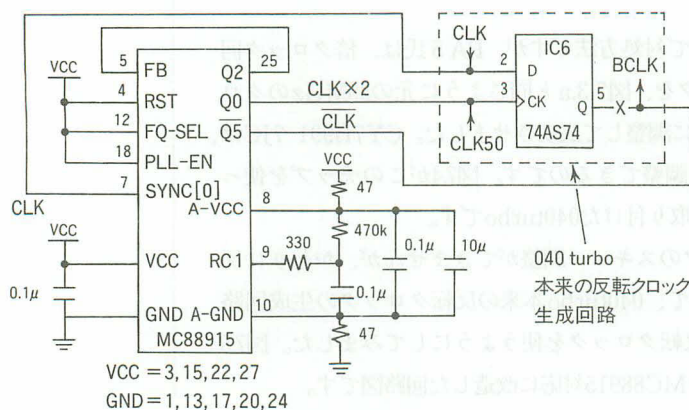



図7.5　040turboの倍クロック回路例2（MC88915の場合）

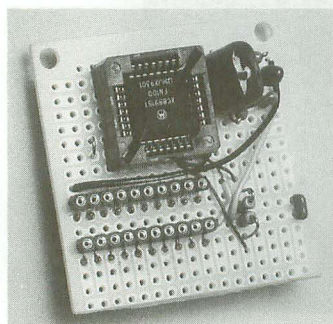



写真7.2　倍クロック回路　(上)アップ
(右)取り付けたところ

禁断の改造技～クロックアップ～

倍クロック回路の直接的なメリットは「取り付けの手間が減る」ということですが、実は別の恩恵もあるのです。倍クロック回路の試みで明らかなように、ICクリップで取り出した50MHzのクロック信号は図7.3.aのように思いっきり歪んでいて三角波のようになっています。しかし、倍クロック回路で作った信号はかなりきれいな波形*1です。この、安定したPCLK信号が作られるということから、「クロックアップ」がやりやすいという恩恵が出てきます。

「VRAMゴミ問題」でわが家のX68030をクロックアップしてみたときは、ICクリップを使うと66MHzのオシレータではなんとか動きましたが、72MHzのオシレータを使うとたまにコケることがありました。もっとも、今は半田付けして直接クロックを取り出しているので、倍クロック回路を使わなくても72MHzで動作させることができますが、半田付けでなくても倍クロック回路を使えば楽々と動作させることができます。さらに、ICクリップによる接続では起動すらしなかった80MHzのオシレータによる68040-40MHzの動作も、なぜかデータキャッシュをオンにすると暴走*2してしまいましたが、データキャッシュなら起動して普通に使うことができました。倍クロック回路のほうがクロックが安定している分、高いクロックが狙えるのです。

＊1
方形波とはいいがたいですが。

＊2
25MHzの68040に40MHzのクロックを与えているのですから、暴走しても当然ですが。

クロックアップは得か？

X68030が相当に速いのは第１章で説明したとおりです。それに040turboを搭載すれば、さらにその２～３倍の速度になりますから、性能的に不満があるわけではありません。そもそもクロックが25MHzから36MHzになっても性能は1.5倍にもなりませんから、本当に高性能を追求するなら、もっと別のアプローチ*3をしたほうがいいでしょう。

それに、クロックアップはメーカーがマージンとして残しておいたタイミング上の余裕を食いつぶすことになるわけですから、安定動作は保証されませんし、半導体素子は動作クロックが上がればより多くの熱を持ちますから、壊れる危険性は高くなります。もちろん、クロックアップのためのマザーボード改造もまた危険な作業の１つですから、これらのリスクを考えると、無理にやるほどのことではありません。

それでも、あえて愛機の性能の限界に挑戦してみるのは、それはそれで意味のあることだと思っています。もっとも、私の場合は「VRAMゴミ問題」の

＊3
安易な道としては、X68を捨てて別のマシンに乗り換えるということです。私としては68060に期待したいところですが、どうなることやら。

対処のためにクロックアップせざるを得なかった*1わけですが、いずれは挑戦してみるつもりでした。

*1 速度というのはすぐに慣れてしまうもので、最近では36MHzの68040の速度にも慣れてしまって、もっと速くならないかと思っています。

クロックアップのマージン

ひと言でクロックアップのマージンといってもいろいろありますが、一番効いてくると思われるのはメモリの応答速度でしょう。図7.6.aのように、DSACKx信号はメモリの応答に先立ってアサートされます。これは、前にも説明したように、68030が①のタイミングでDSACKx信号を認識した後、1クロック後の②のタイミングでデータを取り込むからです。ですから、データが出てくるのは、②よりも一定時間*2前であれば、①より後でもかまわないのです。

*2 「セットアップタイム」と呼ばれます。68030-25MHzの場合は2nsecです。

では、クロックアップするとどうなるのでしょうか。図7.6.bがクロックアップした状況です。①からどれくらい遅れてデータが出てくるかはメモリチップの応答速度で決まりますので変わりませんが、1クロックあたりの周期が短くなるので、結果的にデータが整ってから②までの間隔が詰まってきます。そして、この間隔がセットアップタイムを満たせなくなる*3と、データを取りこぼします。

*3 データシートのセットアップタイムにもマージンがかかっているので、それより多少短くても動くでしょう。

結局、これがクロックアップの限界となります。なお、ウェイトを入れて、図7.6.bの②ではなく、次のクロックの立ち上がりタイミングでデータを取るようにすれば余裕を持ってアクセスすることができます。これが、「VRAMゴミ問題」の対処といっしょに新版GALに盛り込んだウェイト挿入の機能*4です。68040は変換回路を経由して応答を返すようにしているので、このような対処が容易にできるのです。68030の応答も変換回路を経由するようになっていれば対処可能ですが、040turboは残念ながら、そうなっていません。*5

*4 もとから旧版GALにはウェイトが挿入されていましたから、正確にはウェイトを削るか残すかを設定によって選択できるようにしたということです。

*5 040turboの製作ではシンプルな回路を心がけたので、68030の信号にはなるべく手をつけないようにしたのです。

（a）25MHz動作　（b）33MHz動作

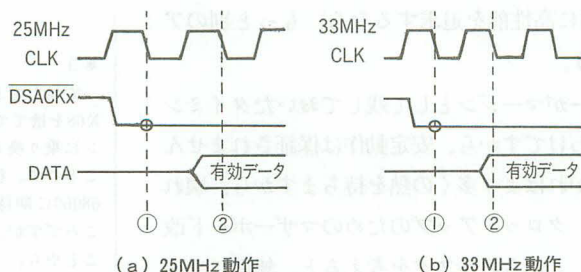



図7.6 DSACKx信号の先出しとメモリの応答速度

クロックアップの動作

私がクロックアップのために施した改造は、単にX68030のプロセッサ用の50MHzのオシレータを外し、オシレータを交換できるようにICソケットにかえただけです。X68030に使われているオシレータは正方形のタイプでしたが、長方形のタイプのほうが入手しやすいので、図7.7のようにソケットを工夫して長方形のオシレータも挿せるようにしています。クロックの切り替え回路[*1]などは使っていません。また、一般にシステムクロックといわれている20MHzのオシレータのほうはそのまま[*2]です。

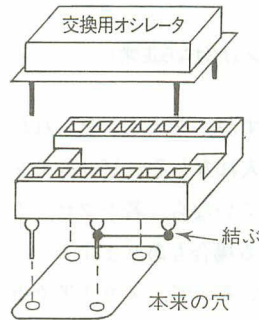



図7.7　オシレータのかわりに挿すICソケット

写真7.3がICソケットに取り換えた様子です。

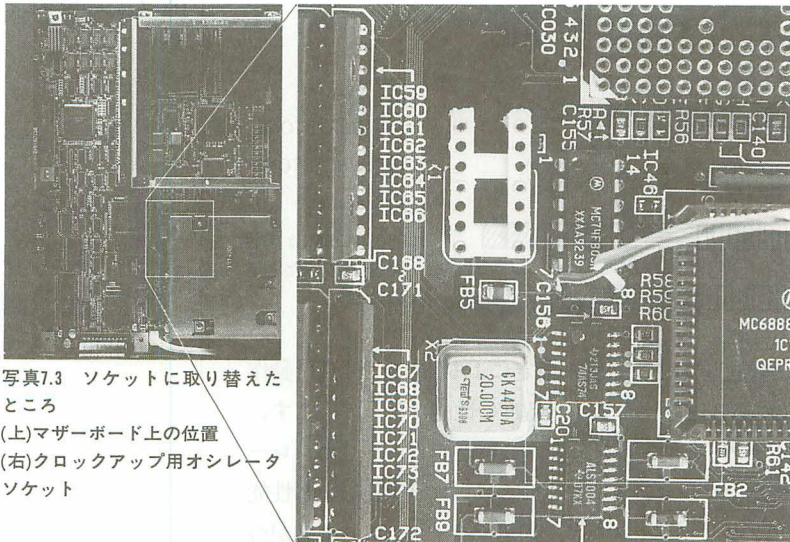



写真7.3　ソケットに取り替えたところ
(上)マザーボード上の位置
(右)クロックアップ用オシレータソケット

＊1
クロックアップの詳しい内容は、「バックアップ活用テクニック」(三才ブックス刊)31号などに出ています。

＊2
X68000互換のI/O系は、プロセッサクロックの25MHzを分周した12.5MHzが使われているようですので、20MHzのオシレータは、I/OスロットのアクセスとSCC（Serial Communication Controller）に供給する5MHzのクロックにしか関係しない気がします。

そして、オシレータをいろいろ交換して試した結果を表7.1に示します。

表7.1　クロックアップの動作状況

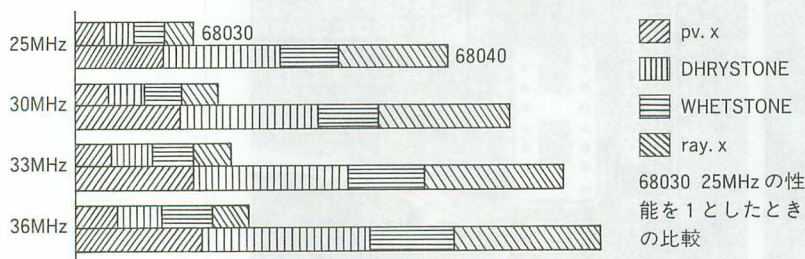
	25MHz	26MHz	30MHz	33MHz	34MHz	35MHz	36MHz	40MHz
68030	○	○	○	△	×	×	×	×
68030 SW1設定*1	○	○	○	○	○	○	△	×
68040	○	○	○	△	×	×	×	×
68040 1ウェイト*2	○	○	○	○	○	○	○	◬

○正常に動作　△起動はするが途中でエラーが発生
×起動せず　◬データキャッシュオンで暴走（命令キャッシュだけなら正常）

ウェイトなしだとさすがに33MHzでもきついのですが、ウェイトを入れればおおむね36MHzでも動作しています。ウェイトを入れるとそれだけメモリアクセスが遅くなるので、へたにウェイトを入れるくらいなら、若干クロックを落としても、ウェイトなしで動かしたほうが得になる場合もあります。

しかし、68040の場合、大きなキャッシュを持っているので、メモリアクセスのウェイトのデメリットがカバーされる可能性があります。

この関係を見るために、pv.xとDHRYSTONE、WHETSTONEの値、そして、レイトレーシングソフト*3について測定してみました。グラフ7.1がその結果です。

グラフ7.1　クロックアップと性能

最もパフォーマンスがよいIC2の1ウェイト設定による68040-36MHz動作は、ノーウェイトの68040-25MHzと比べて1.4倍程度の性能になっています。36÷25=1.44ですから、ほぼクロックアップ分とみていいでしょう。オシレータを取り替えるだけの単純なクロックアップですから、クロック比以上に性能が上がることはあり得ず、メモリアクセスなど、周辺デバイスのアクセスがついていかないため、むしろクロック比以下になるのが普通です。かろうじてクロック比並みの性能が達成できているのはやはりキャッシュでカバーされてい

＊1
X68030マザーボード上のスタティックカラムオフの設定。

＊2
040turbo独自のIC2の設定によるウェイト。

＊3
第1章で出てきたHat氏のプログラムです。

るからでしょう。

68030から68040になったことで性能が2～3倍になったのと比べると、クロックアップは小幅のチューンナップという感じです。まあ、マザーボードに手をかける度胸と技術さえあれば、後はオシレータ代の数千円の出費ですむのですから、コストパフォーマンスの面から比較すると、クロックアップも捨てたものではないかなと思います。ただ、マザーボードを壊したりすると、一気に逆転して高い買い物になりますから、その辺も十分考慮する必要があるでしょう。

禁断の改造技～ハイレゾ～

X68000時代からクロックアップは行われていましたので、結構多くの人がやっているのでしょうが、このハイレゾ改造は、さすがにほとんど聞いたことがない*1ように思います。

ハイレゾ*2表示といっても、X68000の768×512ドットという解像度もハイレゾと呼ばれていましたから、何を基準にハイレゾというかにもよりますが、ここでは、とりあえず1024×768ドットクラスの解像度を「ハイレゾ」と呼ぶことにします。

さて、X68000のVRAMは仮想画面として1024×1024ドットの領域を持っており、画面表示の隠しモードとして1024×848ドット*3の表示機能を持っていました。もっとも、水平同期周波数24kHzでインタレース表示を使って無理やり表示していますのでチラツキが激しく*4、とてもまともに使える代物ではありませんが、それでもハイレゾへの布石ではあったわけです。

X68000でここまでやっているのですから、後継機種のX68030なら、当然、ちゃんとしたハイレゾ、すなわち、ノンインタレースで1024×848ドットの解像度を実現してくれるものと思っていました。

ところが、X68030の表示系はほとんど機能拡張されなかったのです。これには少なからずガッカリ*5しました。

初代X68000から拡張されたのは、X68000Compactで搭載されるようになったVGAモード*6のみです。

もしかしたら、SVGAモードがあるんじゃないか。 *7

＊1
前にNIFTY-Serveで私が報告したことがありますが。

＊2
High Resolutionの略。

＊3
画面モード18です。

＊4
チラツキの目立たない長残光タイプのディスプレイを使っている人がいるという話を聞いたことがあります。

＊5
フルカラー化はVRAMの容量を増やさなければならないから難しいとしても、ハイレゾのほうは比較的簡単にできるのではないかと思います。

＊6
IBM PC/AT互換機の世界で主流の、640×480ドットの解像度の画面モードです。

＊7
こちらはSuperVGAモード、すなわち、VGAの640×480ドットの解像度を超える画面モードです。

とひそかに期待したのですが、それもありません。『X68030 Inside/Out』(ソフトバンク刊)で公表されたX68030の回路図を眺めてみても、やはり、ハイレゾへのハードウェア的な対応はなされていないようです。

何が足りない?

ところで、1024×848ドットがインタレース表示になってしまうのはなぜでしょうか。そして、ノンインタレースにするにはどうすればいいのでしょうか。

これには、ドットクロックが関係してきます。

そもそも画面表示というのは、画面上を電子ビームを左右に振りながら上から下へなぞること[*1]で実現しています。そして、図7.8.aのように、この画面走査のタイミングにあわせてVRAMの情報を順次読み出していき、電子ビームの出力を調整すれば、VRAMの情報に対応した画面ができあがるわけです。

*1
これを「走査」といいます。

そして、このVRAMを読み出すタイミングをとるものが「ドットクロック」と呼ばれるクロックです。図7.8.bのように、ドットクロックが遅ければVRAMを読み出す速度がゆっくりになり、相対的に画面表示の解像度が低くなりますし、図7.8.cのようにドットクロックが速くなれば解像度が高くなります。

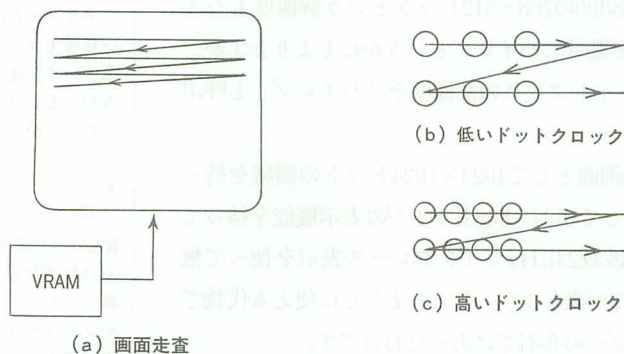



図7.8　画面表示とドットクロック

さて、X68030を見てみると、ドットクロックの元となるオシレータとして69.551MHzと50.350MHzと38.864MHzの3つのオシレータを搭載していることがわかります。これを、OSCIAN2というICで2分周したものをドットクロックとして使っています。最も速い69.551MHzでも2分周した34.7MHzがドットクロックとなり、これで実現できる解像度は768×512ドットです。CRTコントローラの設定値を調整して無理やり表示範囲を広げても、せいぜい、800×600ドットがいいところです。

もっと速いドットクロックにすれば、当然、解像度は上げられるはず。[*2]

*2
むやみにドットクロックを上げても、VRAMアクセスがまにあわなければしかたがありませんが、X68030はデュアルポートRAMを使っているので、かなりの高速アクセスに堪えられそうです。

VGAモードはほとんど使いませんので、VGAモード用の50.350MHzのオシレータを100MHzのオシレータに取り替えてみることにしました。これでも２分周すると50MHzになってしまうので、ドットクロックとしては決して速くはありません。IBM PC/AT互換機の世界のSVGAで、1024×765ドットといった解像度を扱う場合は、普通、75MHzや80MHzといったドットクロックを使うので、２分周されてしまうX68030でそのくらいのドットクロックを作るには本当は150MHzくらいのオシレータをつけたいところですが、100MHzを超えるオシレータは売っていない[*1]ので、しかたがありません。

＊1
特注すれば作ってくれるかもしれませんが。

オシレータの取り替え

ドットクロックのオシレータの取り替えも、クロックアップのためのオシレータの取り替えと手順は同じです。写真7.4のように50.350MHzのオシレータを外して、ソケットを取り付けました。

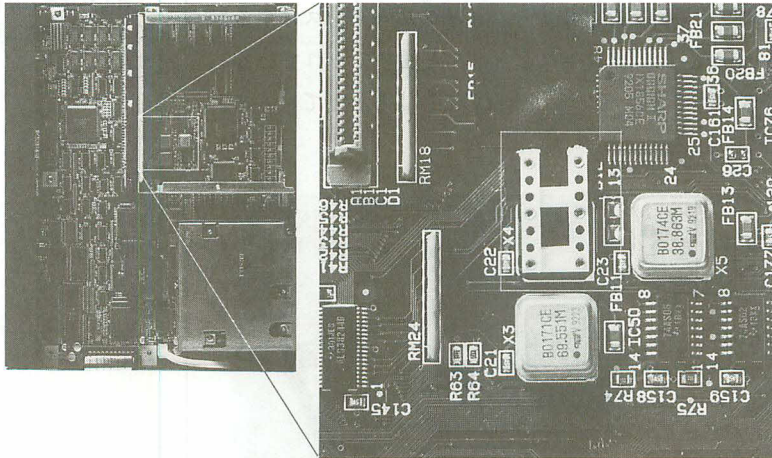



写真7.4　ドットクロックオシレータを外してソケットを取り付けたところ

さて、このソケットに100MHzのオシレータを取り付けるわけですが、そのまま取り付けただけではうまくいきませんでした。画面がグチャグチャになってしまったのです。100MHzのオシレータを外せば正常になります。

調べてみると、X68030の、69.551MHzと50.350MHzのオシレータの切り替えは、図7.9.aのように２つのオシレータの出力をつないでおいて、図7.9.bのHR信号によってオシレータの１番ピンを制御して、片方のオシレータ出力をハイインピーダンスにして、もう一方のクロックしか出さないようにするということで実現しています。ところが、買ってきた100MHzのオシレータは１番ピンの制御ではハイインピーダンスになりません。このため、通常使う69.

551MHzだけでなく、100MHzのほうのオシレータも動いてしまって、ドットクロックがメチャクチャになっていたというわけです。

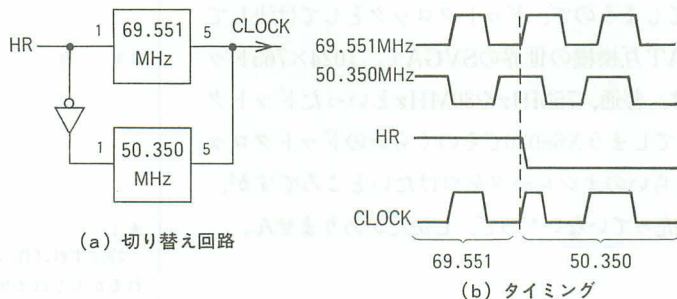



図7.9　ドットクロックオシレータの切り替え

それではと、1番ピンの制御でオシレータ出力をハイインピーダンスにできる品種を探してみると、100MHzも出せるものは残念ながらありません。[*1] しかたがないので、図7.10のように外部回路で出力をハイインピーダンスにするようにしました。74F126が100MHzのクロックをちゃんと通してくれているのかどうかははなはだ疑問ですが、動いているようなので、よしとしています。

写真7.5が、その外観です。

＊1
これも特注すれば作ってもらえるかもしれませんが。

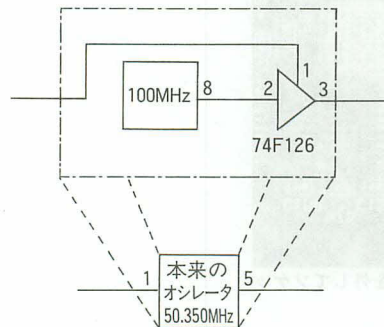



図7.10　外部回路によるハイインピーダンス制御回路

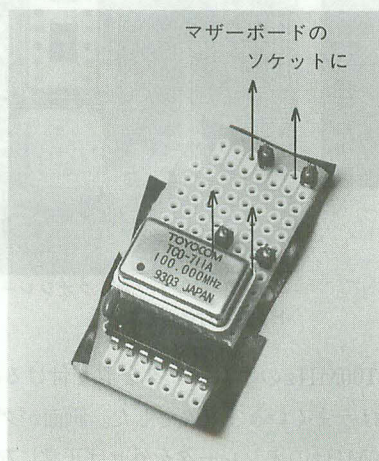



写真7.5　100MHz回路

ハイレゾ表示

実はわが家のディスプレイは、長らく愛用してきたCZ-600Dの調子が悪くなってきたので、思いきって17インチのマルチスキャンディスプレイMAG-17Sというディスプレイに買い替えました。このディスプレイは数年前から発売されているIBM PC/AT互換機およびMacintosh用の有名なディスプレイの１つで、トリニトロンブラウン管を使った高級品です。*1前から憧れていたのですが、X68030の水平同期周波数31kHzのモードなら追従しますが、24kHzや15kHzには対応していないので、買おうかどうか躊躇していました。しかし、最近、急激に安くなったために、店頭で価格表示を見たときほとんどその場の勢いで買ってしまったのです。もちろん、その時点ではハイレゾ表示ができるかどうかもわかっていませんでした。

そのMAG-17Sですが、さすがに高性能です。かなり無茶な設定をしても、なんとか追従してくれます。X68030のCRTコントローラを直接叩いていろいろ試したところ、図7.11のように、水平同期、垂直同期のそれぞれの信号のタイミングを設定することで、1024×768ドットの画面を表示*2させることになんとか成功しました。

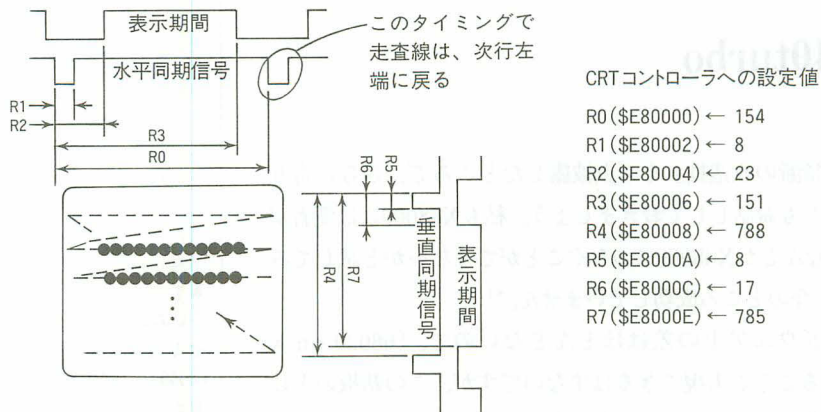



図7.11　同期信号とCRTコントローラの設定

写真7.6が、MAG-17Sにハイレゾ表示させた状態です。このハイレゾ画面で見ると、SX-WINDOWもまったく違った印象になります。もちろん、ノンインタレースですから、24kHzのインタレースによるハイレゾのようなチラツキ*3はありません。

この程度の改造で、まがりなりにもハイレゾ表示ができるようになるのです。もちろんX68シリーズのディスプレイテレビでは表示できませんので、その点

＊1
NANAOのディスプレイほど超高級ではありませんが。

＊2
MAG-17Sの特徴の１つである、画面モードを示す液晶表示パネルでは『SVGA800×600』という判定結果でした。ドットクロック75MHzくらいにしないと『SVGA1024×768』という表示はおがめないのかもしれません。

＊3
もっともドットクロックが遅いので、画面をじっと見つめていると若干チラついているかなという感じは受けます。

も考えると簡単とはいえないかもしれませんが、X68030が標準でハイレゾ表示機能を持っていたら、X68030の評価もだいぶ変わっていたかもしれません。

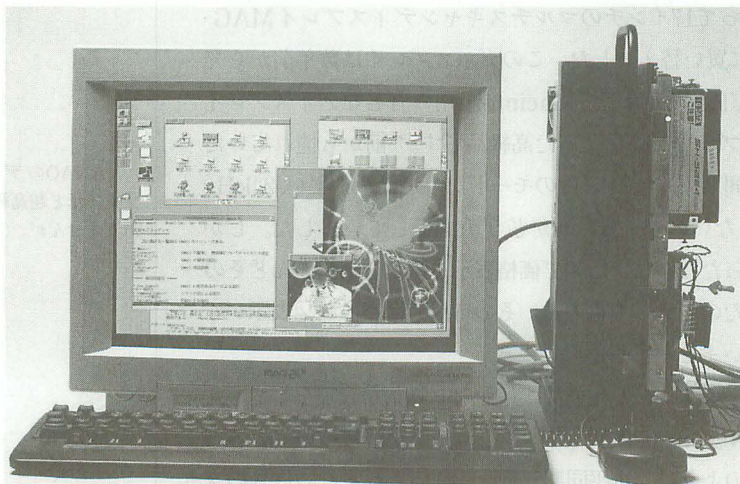



写真7.6　ハイレゾ表示のX68030

X68000と040turbo

さて、X68030側の「禁断の改造技」を2点披露したところで、さらに寄り道をしてX68000についてもお話ししておきましょう。私もX68000には愛着があるので、040turboをなんとかX68000につなぐことができないかと試してみましたが、残念ながら、今のところ成功していません。*1

68020と68030のハードウェア上の差はほとんどないので、「68020 on X68000」の機能拡張をすることで実現できるはずなのですが、この基板の上に040turboをつないだ状態では「エラーが発生しました、リセットしてください」というメッセージが出るところまではいきますが、まともに起動できません*2でした。

それではというわけで、040turboの変換回路自体をX68000対応に改造してみたのですが、残念ながら、こちらもうまくいっていません。もっとも、こちらはマザーボード上の68000のソケットから20cmくらいのケーブルをつないで信号線を取り出すようにしたので、物理的な信号伝達がうまくいっていなかったのかもしれません。

しかし、仮にハードウェア的に040turboをX68000に接続できたとしても性

＊1
いろいろ無茶なことをしたので、最近は68000での動作さえもちょっと怪しくなってきています。

＊2
もともと「68020 on X68000」の68020の動作も、起動はできますが、しばらく使っていると暴走してしまったので、不安定だったことは確かです。

能的にはあまり期待できません。第1章でも説明しましたが、X68030がちゃんと32ビットバスになって25MHzの高速性能を生かせるメモリシステムを持っていたから、比較的シンプルな040turboでもそれなりの性能を出せたのです。16ビットバスの10MHzの68000を想定したX68000のメモリシステムは、68040の動作の大きな足カセとなります。本格的に性能を追求するなら、X68000のメモリとは独立して68040のボード上に32ビットバス幅のローカルメモリを搭載することが必須となるでしょう。

もっとも68040はキャッシュが大きいので、現状の040turboでも「68020　on X68000」のように数十パーセントしかアップしないといった情けないことにはならないでしょうから、「68040を使ってみたい、ついでに少し速くなればラッキー」というくらいのつもりで、試してみるのもよいかもしれません。

ところでもう1点、X68000に040turboを搭載する場合、大きな落とし穴があります。それは、X68000のROM内のルーチンがキャッシュ制御には関知しないということです。

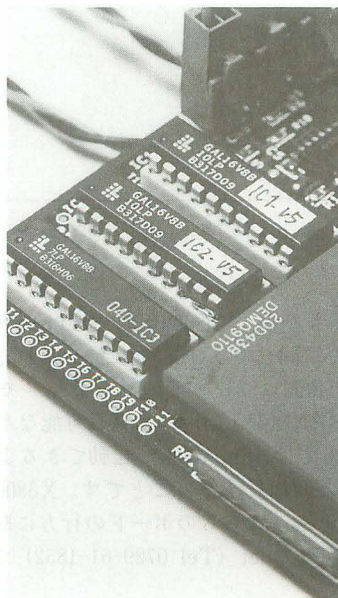
幸い、Human68k ver 3は自分自身でプロセッサをチェック*1して、キャッシュを持っているプロセッサに対してはキャッシュ制御を行うようになって

＊1
なぜか「68020 on X68000」は、プロセッサタイプ3、つまり、68030と認識されていますが。

COLUMN

040turboの拡張性

基板設計段階では拡張用のコネクタをつけることも考えていましたが、アドレスバス32本、データバス32本に、さらに制御系の線も数十本は必要になるということで、ただでさえこみあっているパターンにこれらのコネクタを取り出すためのパターンを加えるのはとても無理ということであきらめました。

ちなみに、040turbo上に未練たらしく並んでいるテストポイントは、拡張コネクタを取り付けた基板を起こすときのためのアートワークの布石です。次の基板を起こす機会があれば、この部分のアートワークをちょっと整理するだけで拡張コネクタが取り付けられるようになるのではないかと思っています。

いますから、Human68k経由で使う分には多くの問題は回避されますが、ROM内のルーチンに関しては040SYSpatch.sysが行うようなパッチレベルの作業ではすまず、キャッシュ制御に関連してくるルーチンをすべて作り替えるくらいのことが必要になってきます。

いずれにしても、簡単にできるかと思っていたX68000への対応なのですが、結構てこずってしまい、今は棚上げ*1状態です。もし意欲のある人は、ぜひ試してみてください。私も、ひと段落ついたら、また試してみるつもりです。

＊1
X68030で動いているものを、わざわざ遅いX68000につなぐ必要はないだろうと投げ出してしまったといったほうが正しいかもしれません。

COLUMN

X68000用68040ボード

040turboを公開してまもなく、X68000用の68040ボードを自作している方から連絡をもらいました。同じように68040に魅せられた人がいたわけです。こちらのボードは、68040側に32ビットのローカルメモリとして、最大でDRAMを16Mバイト、SRAMを１Mバイト搭載可能な本格的なシステムです。すでにこのシステムではHuman68kが起動できるようになっており、現在は安定動作のための調整段階とのことです。X68000ユーザーは、040turboを無理に載せるよりも、こちらのボードの行方に期待したほうが得策かもしれません。現在、FTZ-net（Tel 0729-61-1852）で作業が進められています。

「X68/040turbo」の今後

さて、X68000はいまだにすねたままですが、倍クロックとクロックアップ、そしてハイレゾ改造はなんとかうまくいきました。040turboとあわせてこれらのチューンナップを施したわが家のX68030は、X68030を超えたマシンと自負しています。

大げさかもしれませんが、X68030をベースにした「X68/040turbo」*1という新しいマシンといえなくもありません。

これまでは68040への思い入れだけでがむしゃらにやってきましたが、この本を書くためにあらためてその過程を振り返ってみて、正直いって自分でもよくここまでこられたなと思っています。

ハードウェアについてはそれなりに心得はありましたが、68040自体ははじめて扱うチップでしたし、X68030のほうの資料もありません。ロジックアナライザで波形を調べながら内部の動作タイミングを推定していくほかありませんでした。そんな状況でしたから、自分でも68040を安定して動くようにできるという*2確信はありませんでした。

ソフトウェアのほうもROM内ルーチンやHuman68kのソースなどはありませんから、みんな独自解析です。Human68kが動くかどうかも賭けのようなものでしたし、ましてやアプリケーションが普通に使えるようになるとはまったく考えていませんでした。

また、040turboのプリント基板の製作を決意したときも、そこにはキャッシュオンにすると暴走するHuman68kと、無理やりキャッシュオンにして半分暴走したような状態で動かしたベンチマークプログラムがあるだけでした。そんな状態にもかかわらず、よく040turboの配布にみんな*3参加してくれたものです。ハードウェアやソフトウェアに対する協力、不具合のレポート、そして多くの励ましがなかったら、個人の自己満足で終わっていたでしょう。

そして、いよいよ040turboも「OEM」の形で製品として扱われることになりました。私からの個人的な配布ではやはり限界がありますし、サポートに不安を感じる人もいるでしょう。本格的に製品として取り扱ってもらうことで、より多くの人が安心して使えるようになることと期待しています。

X68030は、X68000時代からの多くのフリーソフトウェアに支えられてきました。これらのフリーソフトウェアが多くの人のレスポンスによって磨かれ、市販のアプリケーションが少なくても、それをカバーする素晴らしい魅力を

＊1
専用ステッカーも作ったので、これを貼れば完璧です。

＊2
「68020 on X68000」は、結局安定動作させることはできませんでした。

＊3
第一次、第二次の配布をあわせると50人を超えます。

X68030に与えています。

同じように、040turboも多くの人に磨かれて、ここまでこぎつけることができました。すでにソフトウェアのほうは完全に私の手を離れていますし、040turboのハードウェアについても、フリーハードウェアとしてX68030を支える魅力的なアイテムの１つになれたらと思っています。

フリーソフトウェアのようにダウンロードする際の電話代程度ではすまないのがハードウェアの痛いところですが、「OEM」に関してはライセンス料などを要求しない[*1]つもりですので、少しは安くなるでしょう。また、回路情報などはフリー（自由）に使ってくださってかまいません。

これが刺激になって、ローカルメモリや二次キャッシュなどの040turboの機能拡張、X68000への68040の搭載、さらには、68060対応など、いろいろなフリーハードウェアがX68の世界を賑わすことを願ってやみません。

＊１
今はまだ設計費の回収が終わっていないので若干の上乗せがありますが。

25MHz
↓
36MHz
倍クロック回路
安定50MHzの
取り出し
ハイレゾも欲しいな

番外奮闘編

patexec.sys

040SYSpatch.sysがシステム全体に対するパッチを受け持つことで、大半のアプリケーションは68040で動作します。しかし、これで万全とはいえません。なかには個別にパッチしなければいけないアプリケーションがあります。

しかし、実行ファイル自体にパッチを当ててしまうと、68040で使うときはいいのですが、今度は68030で使うときに困ります。68040用と68030用の実行ファイルを別にすればいいのですが、わずかなパッチ箇所のために、別のファイルにしておくのはもったいないことです。

そこで登場するのが、patexec.sysです。ダイナミックパッチャと呼ばれるこのプログラムは、別に用意したパッチデータをもとにプログラム実行時に必要なパッチを当ててくれますから、68040用の実行ファイルを用意しておく必要がありません。もともとは、040turboのために作られたわけではないのですが、使用目的を考えると、まったく040turboのためにあるようなソフトウェアといっていいでしょう。

［BEEPs］

中村ちゃぷに❖NIFTY-Serve : GBA02750

patexec.sysって安易なネーミングですね。実は「patch for DOS EXEC」の略で、起動時にMPUの種別を判別して、それに応じてアプリケーションにパッチを施すドライバとして開発したものです。もともとは1993年3月、X68030が登場してすぐにリリースしたもので、X68030対応への過渡期にはそれなりに役に立つのではないかと思い、発表したのですが、案外、いろいろなところで役に立っているようです。

そう。思いもよらぬところで役に立ってしまった……。それは、BEEPsさんの陰謀？

patexec.sys陰謀説

いつのことでしたでしょうか。040turboプロジェクトが本格的に走りはじめて、とりあえずいくつかのアプリケーションが安定して動作するようになった頃のことだと思います。その時点ではまだ040turboの量産配布は行われて

おらず、主にBEEPsさんの手元であれこれ試みておられました。

なぜかすでにその時点では、その陰謀の力が私（中村）にも及んでいたらしく、040turbo基板の量産試作版[*1]が私の手元にありました。

BEEPs：ちゃぷが作ったpatexecって、040にも対応できないかなあ？
ちゃぷ：つまり、040専用にパッチを施すってこと？
BEEPs：そう。今のところ、基板に切り替えスイッチがついていて、030モードでも040モードでも起動するようになっている。MPUを判別して、040モードのときに040専用のパッチを当てられないかと。
ちゃぷ：（中略）ラジャー。

（中略）のところでどんな取り引きがあったのかは定かではありませんが、すでに040turbo量産試作版を使っていたというのがいちばん大きな理由で引き受けました。

その後、patexec.sysはコピーバック[*2]対応にともなって多少バージョンが上がりましたが、とりあえず大過なく、安定動作しているようです。[*3]

なぜ、patexecが必要だったか

基本的に、030と040はユーザーレベルでは完全互換となっておりますが、X68kシリーズにおいてはスーパーバイザレベルで悪の限り（笑）を尽くしているアプリケーションが多く、そういう状態においては030と040では動作が違う部分もあります。いちばん顕著なのがオンチップキャッシュの制御関連ですが、そのほかにもMMUの扱いが変わっていたり、浮動小数点ユニット関連でアヤしい部分があったり、MPU特有の動作[*4]などがあったりします。

patexecのメカニズム

ダイナミックパッチャと前振りがついているpatexec.sys、何をやっているドライバなのでしょうか？　簡単に説明しますと、アプリケーションがファイルから読み込まれて実行される前に、環境に応じてパッチを選んで当てるという動作を自動的に行うものです。実行ファイルに直接パッチを行えない、もしくは行いたくない場合に有効です。もちろん、後からアプリケーションがバージョンアップしたときにはパッチを当てることができなくなりますが、その場合にはパッチそのものが無視されるようになっています。

＊1
90度回さないと基板が装着できない奴ですね。

＊2
040オンチップキャッシュの強力な機能。メモリライトをキャッシングできるので、遅いメモリを使っていてもMPUのパフォーマンスをかなり引き出すことができます。当初040turboは、コピーバックモードに対応していませんでした。

＊3
しかし、デバイスドライバのくいあわせが悪いとき、まっ先に疑われるのはpatexec.sysだったりします。

＊4
あえてバグとはいいません。

パッチは主にバイト単位の書き換えによって行われます。パッチデータは、

```
FC.X /B ＜旧ファイル＞ ＜新ファイル＞
```

で出力されるものと同じ形式をしております。

具体的には、

```
AAAAAA: BB CC
```

という構造をとっており、「オフセットアドレスがAAAAAAで、旧データがBBになっているものをCCに置き換える」と読みます。patexec.sysがパッチを当てる際には、「旧データがBBになっていることを確認して、CCを書く」という動作を行い、もし途中でパッチに失敗したら、「すべて元に戻す」という動作（リバースパッチ）を行うことによって、安全にパッチを解除することができます。

たとえば、foo.xというファイルに対するパッチデータが、MPUが040のとき[*1]、あるいはそれ以外のときに、

```
-4-foo.x
00009E: 61  49
-foo.x
00009E: 61  4E
```

というように2種類用意されることもあるわけで、このときはpatexec.sysがMPUを自動判別して、それぞれに応じたパッチを選ぶことができるようになっているのです。

patexecとlzxと私

私はlzxを常用しています。なぜ……って、ディスクキャッシュにあまり負荷を与えない[*2]のが好きなんです。ところが、X68030とlzxの相性はよいとはいえないものがありました。そう。命令キャッシュです。

もともと、モトローラの68000シリーズでは、命令（プログラム）とデータを別のものとして考えていたようですが、68000ではそれほど厳密に区別しないでもさほど影響はないようでした。しかし、それ以降、MMUが導入されたり、命令キャッシュが設けられたりして、命令とデータの区別をいっそう厳

＊1
040のときにはプログラム名の前に〝-4〟をつけます。

＊2
ファイルのサイズが小さいほど、ディスクキャッシュのヒット率が上がります。

しく行わねばならなくなりました。

現在最もメジャーなインテルのx86系のMPUでは、資産継承のために命令の自己書き換えを許しています。しかし、モトローラのチップでは基本的にそれを認めていないのです。なぜならば、データを書く場所はデータエリアという前提があるからです。

lzxの動作は、「自己書き換え」にほかならず、動作はおおむね次のとおりです。

（1）lzxの自己展開ルーチンをメモリの最後尾に移動（転送）する
（2）移動させられたルーチンが展開を始める
（3）展開されたメインプログラムに制御を戻す

この場合、問題は2つあります。1つは、lzx化されたプログラムの先頭十数バイト（1）が命令キャッシュに残っていること、もう1つは、（2）によって、命令キャッシュに残っている（1）の命令列と矛盾を引き起こすことです。小さなプログラムの場合、lzxに対する命令キャッシュの影響が顕著になります。

patexec.sysは、lzx化されたファイルに対してもパッチを当てることができるようになっています。（2）と（3）の間にpatexec.sysの処理をはさむようにしているのです。このときに命令キャッシュのフラッシュを行うようにしたのですが、それが後に040対応するときに役に立ったのはいうまでもありません。

補足すると、lzx化されたファイルには、（1）、（2）のみを行うエントリ(_LZLoadという内部名[*1]がついている）があり、patexec.sysはそのエントリを呼んでいるだけなのです。

ところが、けっこう後になってわかったことなのですが、最新のlzxで生成されたHUPAIR-LZX[*2]では、_LZLoadのエントリが削除されてしまったのです。しかたがないので、HUPAIR-LZXのときはlzxの自己展開ルーチン内にある_LZLoadエントリに相当する部分をそのまま呼ぶような仕様にしてしまいました。

patexecとlzxと040と私

040になると、それはもう大変です。命令キャッシュ、データキャッシュともに4Kバイトもありますので、下手をすると、たいていのプログラムにおい

＊1
もともとはlzxloaderというデバイスドライバをlzx化するためのツール用に使用されていたもの。

＊2
HUPAIR は板垣氏の提唱した実行ファイル規格で、その実行ファイルヘッダ形式は従来のlzx形式と相容れない仕様でした。

て1zxの自己展開が失敗してしまうことになります。patexec.sysには、もともと030のキャッシュのために、(030専用の) キャッシュフラッシュ処理を入れてあったのですが、これを040turboに対応させるために少々手を加えて、とりあえずはよしとしました。

ところが、やがてコピーバックモードが安定してきました。すると、別の問題が発生しはじめました。68040の特権命令にはキャッシュのメンテナンスを行う命令があり、それらには「命令キャッシュを／データキャッシュを」「消し去る(cinv)／メモリに書き戻す(cpush)」という動作モードがあるのです。しかし、コピーバックのことを考えていなかった私は、patexec.sysのキャッシュフラッシュ時にcinvを行っていました。それに気づいて、あわててcpushに書き換えました。

そういうわけで、現在のところ、040turbo用patexec.sysのバージョンは、0.2です。

私は欲望のかたまり

たいした野望もなく書いたpatexec.sysは、このようにして040turboプロジェクトのなかに引き込まれてしまいました。いやあ、思いもかけぬところで役に立つこともあるもんですね。

ただ、とりあえずは動いているのですが、まだまだ改良する余地はありそうです。ちょっと考えてみただけでも、

・パッチに失敗したことがわからない。パッチに失敗すると何も起きない(何もなかったことになってしまう) ので、たいていはそのまま暴走してしまう。このとき、白窓を出して中止を促すことができるようにしたほうがいいのではないだろうか？

・patexec.sysには、起動時にパッチ情報の入ったファイルを与えるようになっているのですが、パッチ情報を追加したらHuman68kを再起動しなくてはならない。

・正確に計測したわけではないのですが、DOS EXECをフックしているためにシステムのパフォーマンスが少なからず低下しているのかもしれない、という危惧がある。幸か不幸か、私は鈍感なので、そこには気づいていないだけなのかもしれないのですが。いちおう、システムの足を引っ張らないようになるべく気を遣っているつもりです。パッチファイル名をハッシング*1してあるだけですが。

＊1
ハッシュ法というアルゴリズムのテーブルを作ること。

さらに考えなければいけないことがあります（ああ、頭が痛いっ）。

前述したように、オンチップキャッシュのフラッシュをcpusha（全キャッシュのフラッシュ）によって行っているのですが、これがまた実行時間のかかる命令でありまして、平気で数百クロック、ワーストケースでは数千クロックくらい食ってしまうことになります。そのようなときには、ライン単位、もしくはページ単位のキャッシュフラッシュを使えばいいのでしょうが、まだそのようなコードを入れていません。これも今後の研究課題の１つです。

X68030ユーザーの生活をがらりと変えてしまった040turbo。こんなおもしろいプロジェクトに少しでもかかわることができたというのは、まことにありがたいことです。私？もう040なしでは生きてゆけない体にされてしまいました！

［参考文献］

MC68030 ユーザーズ・マニュアル（モトローラ）
MC68040 ユーザーズ・マニュアル（モトローラ）
MC68040 Designer's Handbook（Motorola USA）

68040用浮動小数点演算パッケージ

FLOAT040.X

68040の浮動小数点演算機能は68882のサブセットとなっています。このため、X68030に添付されている68882対応の浮動小数点演算パッケージFLAOT4.Xを使うことができません。しようがないので、040turboの開発中はずっとソフトウェアエミュレーションのFLAOT2.Xを使っていたわけですが、せっかくの68040が台無しです。もったいないなあと思いつつも浮動小数点演算の世界は門外漢なので、自分で68040用のFLOATパッケージを起こすことなど到底無理だし困ったなあと思っていたら、この思いが通じたのか、発表されたのがFLOAT040.Xだったのです。

［BEEPs］

鈴木　国文❖NIFTY-Serve : GBH00172

About FLOAT040.X for 040turbo on X68030...

当初、こんなものを大々的に発表するはずではなかったのです。

MC68040になったら、今までの浮動小数点ドライバじゃなぁ〜。誰か作るのかな……。

といった感じで、勝手にX68030上でシコシコ浮動小数点演算ドライバを作ってたわけです。

それがなんの間違いか、今では「例外処理の修正バージョンをアップして……」などとNIFTY-Serveへ書くはめになってしまいました。もちろん、理由なんぞわかりません。

実は私も040アクセラレータなるものを考えておりまして、ギジツ系会社員の特権を利用してマニュアルを取り寄せたり、*1「業務」と称してMC680x0やコプロのマニュアルとにらめっこしてたり……。

040turboを目にしたのはMMNetの書き込みで、それを読んだ日に参加を申し込み、年末に買う予定だったX68030を８月に繰り上げた次第です。当時はX68030も品切れで、なかなか入荷せずに待たされた記憶があります。*2

それ以来、FLOAT040.Xへ執着してしまい、数値演算の本を探しに本屋を

＊１
たった電話１本で10冊近いマニュアルが届いたときにはさすがにビビりましたが。

＊２
やっと届いたのが夏期休暇の直前だったので、予約していた新幹線のチケットが……。でも、結局、帰りましたがね。

渡り歩いたり、会社の研究部署へ行き、参考資料を探し回っていたり、以前に買ったMacintoshのMC68040アクセラレータの解析を仕事中にやってたり……。[*1]

そろそろFLOAT040.Xの話をしましょう。

ver1.10で表記しましたが、これは私のオリジナルではありません。FLOAT040.XはFLOAT2.Xを逆アセンブルしてパッチを当てています。たとえば、もともとあった倍精度加算のルーチンの頭にFLOAT040.X独自のルーチンをくっつけて、倍精度加算が呼び出されたら独自のルーチンへ、他の関数から呼び出された場合はその関数自体は変更していないので残してあったFLOAT2.Xの加算へジャンプする、といった感じです。このため、加算そのものは速くなってますが、加算を必要とする関数は速くなっていません。他の浮動小数点ドライバと比べてサイズが大きくなっているのは、このためです。将来的にはすべてオリジナルで提供するつもりですので、もっとスッキリしているでしょう。

FLOAT040.Xで主に速くなった部分は、倍精度の四則演算、平方根の計算、各種精度の変換[*2]です。ご存じの方も多いでしょう。MC68040のFPUはMC68881/2のサブセットとなっており、指数・対数・三角関数などといった、比較的使用頻度の低いものははずされています。実際にどの程度までスピードアップになったのかというと、ノーマルX68030+MC68882と比較して3倍程度となっています。ただし、浮動小数点数演算をガシガシ使うソフトの場合です。もちろん、他のソフトでも多少の効果はあります。

具体的には、単純ループで3倍前後、レイトレソフトMirageを使用した場合で計算開始から表示まで1：11：47かかっていたのが、0：27：33になりました（バンザーいっ!!）。当然のことながら、これにはFLOAT040.Xだけではなく、040turboの効果も含まれています。

β版では各種関数も高速化するはずだったのですが、参考にした資料の解説が不適切で、計算結果が保証される範囲についてまったく記述されていなかったのです。指数関数や三角関数が無制限にxの式で表せるとは思ってはいませんでしたが……。

結局、頭の中がゴチャゴチャになってしまい、ついでにソースもグチャグチャになり、はじめからやり直して発表したものが現在のシリーズとなっています。

当然ながら、はじめからパーフェクトなものができるわけではありませんので、今は各種の例外処理と各種関数について書いています。

実は例外処理をやってなかったことが利用者の手によって発覚してしまい、

＊1
会社で使用しているMacはIIciで、Daystar社の40MHz Turbo040を取り付けています。

＊2
最新版のver1.2は、倍精度のsinとcosもエミュレートさせています。

デバッグによる徹夜続きと、「2週間で製品を作れ」との会社からのお達しの二重のストレスで神経性の胃炎を起こしてしまいました。それでなくても、タバコの本数は増えるわ、空き缶やペットボトルは散乱してるわ……、結構地獄を味わいました。

朝10時に出社し、退社が翌日の午前2時〜3時、その後でデバッグですから、寝る頃にはウゴウゴルーガが終わっている始末です

話が脱線したので戻しましょう。

FLOAT040.Xの不都合に関してですが、浮動小数点の演算でオーバーフローとアンダーフローになると誤動作する恐れがあります。それと非正規化数の処理も若干怪しいです。「非正規化数」というのは、指数部分が0で仮数部分が0以外の数値表現のことで、アンダーフローぎりぎりの数字を表現しているものです。MC68040の場合、この数値表現はサポートされておらず、この形式を使うとMC68030ではリザーブとなっている未実装データタイプ例外ベクタ$37へ飛び込みます。パック形式も同じベクタへ飛び込みます。FLOAT040.Xは指数に強引に1を加えてますので、元とは違う値になってしまいます。具体的な問題としては、レイトレソフトなどで正常に計算できなくなるかもしれません。今のところ、支障はないようですが……。

私がチェックしたところ、画像圧縮やエフェクトでは大丈夫なようです。これらの例外処理に関しては「エイヤっ！」で処理を行わせていますが、ハングアップやエラーメッセージ（例の白窓）が表示されることは防いであります。

また、普通ならFLOAT040.Xが呼び出された時点でコプロのレジスタの退避をさせるのですが、現在はまったく行っておりません。そのため、直接コプロを扱うプログラムは誤動作したり、レジスタにゴミが残ってしまいます。さらに、moduloの計算の都合上、精度の丸め処理を0へ近づけるように設定させています。これに関しては、moduloの演算後に該当フラグをクリアすればいいだけですが、手抜きをしていて現在は行っていません。*1

＊1
FLOAT2.Xは丸め処理を近似値にもっていくようになっています。

FLOAT040.Xの使用方法には特にオプションはないので、そのままconfig.sysか、autoexec.batに「FLOAT040.X」と書くだけでかまいません。それと、BEEPs氏の提案で、FLOAT040.Xの次に別のドライバを指定すると、040turboを使用しないときにはそちらへ移ります。たとえば、config.sysに、

```
DEVICE=..¥FLOAT040.X
DEVICE=..¥FLPAT2.X
           ↑ここは使用する環境にあわせます。
```

と記述して040turboを使用すると、FLOAT040.Xが起動され、次の行に記述されたFLOAT2.Xは「すでに起動されています。」と表示され、常駐を中止します。040turboを使用しないときは「FLOAT040.X：MC68040ではありません。Quit Program...」と表示され、FLOAT040.Xの組み込みをせずに終了し、その次の行に書かれたFLOAT2.Xが動き組み込まれます。

また、autoexec.batに、

```
..¥FLOAT040.X
..¥FLOAT2.X
```

と書いた場合にも同じような動作を行います。

この場合、040turboを使用しないときには、FLOAT040.Xは〝－1〟を返し、常駐せずに終了します。ここで何を行っているかというと、MPUの種類やMMU、FPUの有無などの情報が書き込まれているアドレス、＄00000CBCの内容をチェックしているのです。

FLOAT040.Xの今後の予定としては、各種関数の高速処理ですね。モトローラは、同一演算に関して同クロックのMC68882＋MC68030よりも速く処理できるといっていますので、やはり、そのとおりになるようにしなければならないでしょう。モトローラの資料では、CORDIC（後述）という方式ではなく、別の方法で演算するほうがよいと書かれています。FLOAT040.Xのver 1.2では、確かにsinの演算速度がMC68030＋FLOAT4.Xの２倍程度になってはいますが、小数点以下７桁目からFLOAT2.Xと異なっている（５次のテーラー展開を使用）のと、040turbo＋FLOAT2.Xよりわずかに遅いといった状況になってしまっています。さすがに、MC68881/２の内部を解析するとか、μROMの資料がほしい、などとはいえません。モトローラにはオリジナルのライブラリ集がありますが、その価格は100万円[*1]とか……。

その後は……その頃にはMC68060が入手可能になってるかな？[*2] X68030に外部キャッシュをつけるのもおもしろいでしょう。でも、「親亀子亀状態」になってしまいますね。後はリザーブになっている＄01000000からのエリアも非常に興味深いです。バスもMacintoshのPDSのようにMPUを交換したり、各種ボードがつけられるような規格を作るのもいいでしょう。

040turboを使いはじめて、おおよそ４ヵ月たつのでしょうか。つねに頭に入れていたことは「キャッシュのヒット率」です。[*3]極力ループ内の処理を小さくまとめてヒット率をガンガン上げることです。MC68040にはせっかく４Kバイト＋４Kバイトのキャッシュがあるのですから、活用しない手はありま

＊1
無料で入手できるライブラリをお持ちの方、あるいは知っておられる方、情報提供お願いします。

＊2
私は知りませんが、MC68060のFPUもサブセットなのでしょうか？

＊3
FLOAT040.Xは、そんなに厳密にCPUのクロック数を計算して……というようなことはしていませんが。

せん。キャッシュをディセーブルした040turboというのはちょっと悲しいものがあります。

それでは最大級の感謝を込めまして、

040turboを作られたBEEPs氏をはじめ、
ドライバ、パッチャーを作られた方々、
アドバイス・アイデアを提供していただいた方々、
バグ情報を提供してくださった方々、
私の拙いプログラムをお使いの方々、

ありがとうございます。これからもよろしくお願いします。

＊CORDIC（COordinate Ritating DIgital Computer）。求めようとする関数や引き数によって決定されたベクトルを、回転させることによって値を算出する方法。電卓に利用されている。また、回転も回転させようとするベクトルと直角になるように行うのでx座標とy座標の入れ替えですむ。

＊5次のテーラー展開

$$y=\sum_{k=0}^{n}\frac{(-1)^{k}\times X^{(2k+1)}}{(2k+1)!}$$

というのが公式、だが無限回計算させるのはナンセンスなので、FLOAT040.Xはk＝4まで打ち切っている。「4」という数字は、FLOAT2.Xとスピードを比較して出てきた値である。

pfloat.x

68040の浮動小数点演算命令は68881のサブセットになっているため、68030＋68881の組み合わせをターゲットに開発されたプログラムを68040で走らせることができません。現時点では、68030＋68881というターゲットをサポートしているのはGCCだけで、XCのほうはサポートしていませんから、市販アプリケーションで困ったことはほとんどありません。[*1]

しかし、もしエラーで実行できなくなった場合、威力を発揮するのがpfloat.xです。これを常駐させておけば、68040が未サポートの浮動小数点演算命令をエミュレートして実行してくれます。

［BEEPs］

＊1
私の知っているかぎり、DōGA-CGAシステムのREND30.Xくらいだと思います。

険しき浮動小数点ドライバへの道
—挫折・挫折・挫折編—

中村ちゃぷに❖NIFTY-Serve : GBA02750

動機——息切れ、目まいのもと

あまり詳しいことは書けませんが、以前、私は仕事で浮動小数点演算にたずさわっていたことがあります。なぜかそのことを知っていた[*2]BEEPsさんに頼まれた[*3]のが事の発端です。

当初の構想では、「新たに浮動小数点パッケージを書きましょう」ということだったのですが、それは無理というものです。業務でさんざん苦しんだものを今度は趣味で……。考えただけでも目の前まっ暗です。それに、そんな時間は私にはとれそうにありませんでした。そもそも、仕事でやっていたものに比べて今回の作業は規模がかなり大きくなることがはじめからわかっていました。

とりあえず、ひととおり機能することを目標として、浮動小数点演算ドライバをなるべく少ない手間で書いてみよう、ということになりました。

＊2
私が吹いて回ったという説があるらしいのですが。

＊3
私が買って出たという噂もあります。まったく自分を何様だと思ってるんでしょうね。

68040の浮動小数点演算

まずは、68040のFPU*1について軽く触れておきましょう。

「MC68040ユーザーズ・マニュアル」には、「68040のFPUは、MC68881/882コンパチブル」である旨が書いてあります。なるほどFPUはコンパチかもしれません。しかし、実際にはそれは68882のサブセットでしかないのです。私たちが浮動小数点と聞いてまっ先に連想するであろう三角関数などは68040のサポート外なのです。三角関数、対数関数などの超越関数、およびPacked decimal*2などは68040単体では処理することができず、専用の浮動小数点演算パッケージを使用してエミュレートしなければならないのです。

マニュアルには、「浮動小数点エミュレータ・パッケージに関しては、最寄りの営業所におたずねください」と書いてあります。ところが、浮動小数点演算パッケージのソースは300万円（?）だそうです。オブジェクトにしてしまえば、コピーは自由だそうですが、X68000用の68040浮動小数点エミュレータなど転がっているわけがありません。*3 これでは040turboに使うわけには

＊1
Floating Point Unit、浮動小数点演算ユニット。

＊2
いわゆるBCD。16進数のA～Fを使用しないことによって10進数との親和性を強めた形式。私の記憶するかぎりではMSXのBasicがこれを採用していました。

＊3
当たり前です。X68040など出ていないのですから。

COLUMN

ソフトウェアによる浮動小数点演算

X68030になってようやくコプロセッサを使用することが可能となり、演算パッケージに頼らなくても実数演算バリバリのコードが書けるようになったと思ったのもつかの間、68040を使う段になって、またもや実数演算パッケージ（FPSP）が必要になってしまいました。まさしく時代逆行モノといってもいいでしょう。

ところが、モトローラの資料によると、「FPSPは68882よりも計算速度が速い」ようなことが書いてあります。これは、いったいどういうことなのでしょう。そんなに68040は高速なのでしょうか？

Yes、68040のFPUからは超越関数の演算を行うためのマイクロコードは削られてしまいましたが、かわりに基本的な演算が速くなったようです。FPUにおける実行時間が加減算で3クロック、乗算で5クロックというから驚きです。なお、除算は37クロック、平方根は100クロック以上かかるようです。68882では、加減算が約20クロック、乗算が約40クロック、除算が約70クロック、平方根は約80クロックだということです（あれ？　平方根はこちらのほうが速いみたい。なぜだ!?）。これらの数値から、68040のパフォーマンスを十分うかがい知ることができるでしょう。

むかしむかし、パソコン（そのころはマイコンと呼ばれていましたね）の8ビットCPUは乗算機能を持ち合わせていませんでした。やがて、乗除算命令を持つCPUが各社より登場してきましたが、それらの命令は一般の命令に比べると遅い命令でした。

いきません。アマチュアは68040を用いたシステムを設計してはいけないのでしょうか。*1

もともとX68000が16ビットだった頃には浮動小数点演算命令はなかったわけで、そういう意味ではX68030で68882専用にコンパイルしたCプログラムくらいしか浮動小数点演算命令を使用していないことになります。それなら、浮動小数点演算命令を無視してしまえば気が楽なはずですが、68040のFPUはそんなふうにして捨ててしまうにはあまりにも惜しい代物です。四則演算にかぎっていえば486など問題にならないくらいに激速なのです。特に乗算は速く、64ビット×64ビットの演算が5クロック前後ですんでしまうのです。後になってわかったことですが、実はこれは32ビット×32ビットの整数乗算よりも速いのです。乗算は実数演算のなかではかなりたくさん出てきますので、もし68040のFPUが実用になれば、きっと笑いが止まらないことでしょう（コラム参照のこと）。

＊1
モトローラにかぎらず、一般的にチップメーカーはエンドユーザーに対しては冷たいものです。

C O L U M N

乗除算がさほど速くないCPUにおいて超越関数を高速に計算するためには、加減算と桁シフトの組み合わせにより実現できるアルゴリズムが有用でした。三角関数ではCORDICという、ベクトルを回転させながらテーブルを引くアルゴリズムがあります。指数対数は、加法定理「logxy＝logx＋logy」の応用で、ひと桁（たいていの場合、1ビット）ごとにテーブルを引くアルゴリズムがあります。68882の計算結果を見るかぎりでは、内部のマイクロコードでは、これらのアルゴリズムで演算を行っているようです。

ところが、乗除算の高速なCPUでは、ビット単位の演算を行うよりもテイラー級数展開による近似式を使用することによって十分な演算速度を得ることができるのです。68040でも、どうやら随所で級数展開による近似式を使用しているようです。「MC68040 Designer's Handbook」によれば、「三角関数は、（68882で使用している）CORDICというアルゴリズムに比べて近似式を使用したアルゴリズムのほうが速いので、そちらを使用している」旨、記述があります。

1つ忘れてはいけないことがあります。ソフトウェアで超越関数の計算を行うことは、これらのようにアルゴリズムが確立していれば、さほど大変ではありません。ソフトウェアで大変なのは、エラーチェック。そして、デバッグです。今回の040turboでは、ソフトウェアによるエミュレートを採用しない方針で作業を進めてきましたが、その理由の1つにはデバッグの大変さ、デバッグのためのマンパワーが明らかに不足しているということがあったのです。

まず、頭の中で挫折

モトローラの供給するFPSPはコストの面からいって今回はあきらめなければなりません。どのようなアプローチでいけば、少ない手間でドライバが作成できるかを考えてみました。

まず、FLOAT2.Xのようなものにかぶせる形でエミュレータを書いてしまうのはどうかと考えてみました。ところが、FLOAT2.Xでは演算精度がたかだか52ビットです。しかし、68040では64ビットの演算精度が要求されます。まあ、趣味でやるものだから、演算精度は少しくらい低くてもかまわない……と思ったのですが、あまり気の進むアイデアではありません。これは頭の中にしまっておくことにしました。

この発展形としては、XCについてくるFLOATEML.L*1をリンクしたドライバを作る、というアプローチもあるでしょう。

誰か似たようなことを考えて、フリー*2の68040用浮動小数点演算パッケージを作ってくれたりしないかな、とも思いました。

残った可能性としては、X68030以前に使用されていた68881ボード（CZ-6BP1/CZ-6BP1A）を使用するしかないかしら。このときばかりは、X68030用の68882が「コプロセッサ」*3であることを呪うしかありませんでした。

＊1
FLOAT2.X の演算部とまったく同じものがライブラリ化されています。

＊2
フリーという言葉にはいろいろな意味がありますが、この場合は使用料が無料、もしくはタダ同然という意味。

＊3
Co-processor；CPUの処理を補佐するような動作を行うプロセッサ。アプリケーションレベルからはCPUと一心同体のものとして扱えます。

可能性を信じて

結論から申しますと、68882をコプロセッサではないものとして扱うことは可能です。

X68000シリーズ用にメーカーから発売された68881ボード(CZ-6BP1/CZ-6BP1A）は、商品名が「数値演算プロセッサボード」であったと記憶しています。一般的にはコプロセッサと呼ばれていたはずの68881は、X68000で使用するときはコプロセッサではなかったのです。コプロセッサとしてではなく、周辺デバイスとして68000に接続することが可能でしたので、「数値演算プロセッサボード」という形での商品化が可能だったわけです。

もともと68020/68030のコプロセッサとして使われることが前提であった68881/68882（以降、68882で統一）は、どのようにしてX68000で使用されているのでしょう。

簡単に説明しておきます。68000には、実際にはアドレス空間が複数存在しているのです。それぞれ「ユーザープログラム空間」「ユーザーデータ空間」「スーパーバイザプログラム空間」「スーパーバイザデータ空間」という名前がつ

いています。X68000のハードウェアでは、これらは同一のメモリに割り振られているのですが、ハードウェアの設計によっては、これらをまったく別の空間に割り振ることができるのです。オペレーティングシステムによる保護機能*1を援助するためです。CPUがメモリをアクセスする際には、必ずアクセスするアドレス空間を指定する信号がいっしょに出力されます。それが、ファンクションコードFC（Function Code）信号で、8通りの状態を表すために、実際には表1のようにFC0、FC1、FC2の3ビットに分かれています。

＊1
しかし、実際にはすべての空間を同一メモリに割り振ってあるマシンがほとんどなのです。

［表1］

FC	FC 2	FC 1	FC 0	アクセス空間
1	0	0	1	ユーザーデータ空間
2	0	1	0	ユーザープログラム空間
5	1	0	1	スーパーバイザデータ空間
6	1	1	0	スーパーバイザプログラム空間
7	1	1	1	CPU空間

68020以降、コプロセッサデバイス用にFC＝7のCPU空間が新設され、コプロセッサはCPU空間に存在するデバイスという位置づけがなされました。このようにすれば、コプロセッサはアプリケーションプログラムからはアクセス不可能でもCPUからはアクセス可能にできるのです。

CPUがコプロセッサ命令を発見すると、まずはCPU空間にあるコプロセッサに命令の実行を要求します。その後、必要に応じてコプロセッサがCPUにアドレス情報、CPUレジスタなどのデータを要求します。CPUとコプロセッサのやりとりはアプリケーションの見えないところで行われます。

「数値演算プロセッサボード」の場合は、ほんとうはアプリケーションが面倒を見ずにすむはずのCPUとコプロセッサのやりとりを、アプリケーションに受け持たせることで実現しています。この場合、68000に接続するのが前提なので、FC＝7のCPU空間を使わずに、アプリケーションからアクセスできるメモリ空間にコプロセッサをぶらさげてあります。CPUから見れば、コプロセッサではなく、単なるI/Oデバイスとなるわけです。

では、68030はCPU空間をどのようにして使って68882をアクセスしている

［表2］

FC	A31			アドレスバス				A 0
111	0000	0000	0000	0010	ccc 0	0000	000R	RRRR

ccc　＝コプロセッサID（68882は通常001）
RRRRR＝CIRレジスタ（コプロセッサにマッピングされているレジスタ）

のでしょうか。「MC68030ユーザーズ・マニュアル」によれば、表２のようになっているようです。つまり、68882はFC＝７のアドレス$00022000〜＄0002201Fを使ってアクセスされていることになります。

FCは、通常は68030が自動的に判断して出力することになっているのですが、実はFCを任意の値にして普通ではアクセスできないアドレス空間にアクセスできる命令があったのです。それがmoves命令です。これを使えば、「数値演算ボード」とほぼ同じ方式で68882をアクセスできます。

前振りが長くなってしまいましたが、

よし。これでいこう。「movesでコプロセッサを直叩きする！」

安直でも動けば勝ちよ

さっそく、BEEPsさんに前項の研究結果について報告しました。演算速度はともかく[*1]、きっちりと動けばいいのであれば例のmovesを使う方法が安直でよいということで、そのことを念頭に置いて浮動小数点演算ドライバまわりの作業を進めていこうということになりました。

まずは手元のX68030で実験することにしました。この実験は安直な方法で行いました。X68030が発売される以前から出回っていたfppp.xというプリプロセッサがありました。これに、fmove.d ....などのコプロセッサ命令のニーモニックを与えると、「数値演算ボード」を直接アクセスするコードに落としてくれるというスグレモノです。このfppp.xが落としてくれたコードを、私が手作業でmovesを使用したコードに変換し、テストプログラムを作りました。予想どおり、何の問題もなく動いてくれました。[*2]

この頃はまだ040turboの量産試作版もできあがっていなかったので、テストプログラムの評価をBEEPsさんに行ってもらいました。最初はきちんと動作しなかったそうですが、やがて正常に動作するようになりました。原因は、コプロセッサインタフェースの信号タイミングが思いのほか厳しく作られていたからだそうです。もともとCPUとコプロセッサはすぐ近くに配置するものなので、タイミングを厳しくしても整合性はとりやすいですからね。

＊1
本音をいえば、「ともかく」ではなく、極限を追求したいところですが、そこまでの体力はありません。

＊2
ただし、アクセス手順を間違えると、数値演算プロセッサボードと違い、エラートラップが発生してしまいます。

果てはXC68040と私

さて、そうこうしているうちに、040turboの量産試作版ができあがりました。浮動小数点演算ドライバを製作するために、試作版をやや早めに手配してもらったのです。

そして、ぼちぼち浮動小数点演算ドライバを書きすすめていきました。名称を「pfloat.x」にしました。pはpseudo*1のpです。モトローラより供給されている浮動小数点エミュレータの名称がFPSPだということで、pfloat.xという命名は「本家FPSPのように振る舞えども、中身はインチキくさい」という意を込めたものです。

機能も非常に制限されたもので、サポートしているものは超越関数など未定義命令だけで、Packed decimal、非正規化数はまったくサポートしていないというシロモノです。とりあえず、Cコンパイラが吐くコードが動いてくれればよいという気持ちで作成しました。

何はともあれpfloat.xができあがったので、さっそく数名の方に試用してもらいました。案の定、「うまく動いてくれない」という声が多く聞かれました。そりゃそうです、私の手元でもバグりまくっているのですから。関数電卓プログラムのようなものを作って動かしているかぎりではなかなかボロが出ません。しかし、グラフを描かせてみると、もう結果はヒサンそのものです。ところどころ、まったく嘘の値を計算してしまっているようです。……というよりは、むしろ、嘘の値が関数に与えられてしまっているようです。

その後、あれこれ試してみたのですが、どうもXC68040のバグではなかろうか、という結論に達しました。68040に内蔵されている四則演算がすでにおかしな値になっているのです。XC68040のバグと決めつけるのは早計かもしれません。もうすぐ出回るともいわれているMC68040を入手してみないことには何ともいえません。

四則演算しか使用していないはずの簡単なテストプログラムでさえ、まともな値を出さないのを見て、しばらくはこのpfloat.xの作業を中断することにしました。そうこうしているうちに、私の本業が忙しくなって、040turboの研究がだんだんできなくなってしまったのです。もともとはpfloat.xを今回の原稿のネタにするつもりでしたが、実験作品のようなものに始終してしまい、あまりカッコウがつきませんでした。

動いているように見えるのならそれでよい、という見方をしていただければ、今回の実験はとりあえず成功（?）です。なにしろ、はじめは実数演算を行っているアプリケーションが途中で「エラー（$000B）が発生しました」*2を起

＊1
［形容詞］似非の。「ぷせうど」と発音してはいけないのですが、心の中ではそう読んでいます。

＊2
1111 lineemulationのエラー。68040にて未定義の浮動小数点演算を実行しようとすると出ます。

こしまくっていたものが、ピタリと収まったのですから。まあ、なかには（内部で正しくない結果が出ているので）無限ループに陥ってしまうようなプログラムもあります。

今回は実験レポートのようなものになってしまいましたが、ようやく仕事のほうが一段落ついた*1ので、pfloat.xの実験の続きを再開したいと思います。NIFTY-Serve FSHARP1にてどしどし研究発表をしていきますので、みなさん課金を恐れず*2、遊びにきてくださいね。FSHARP 1の回し者ではありませんが、いつもお世話になっていますので、ちょっとだけ宣伝ということで……。

＊1
これを執筆している3/23現在、「無職」です。

＊2
リアルタイム会議（チャット）さえ行わなければ、さほど課金は高くなりません。

［参考文献］

・MC68030 ユーザーズ・マニュアル（日本モトローラ）
・MC68040 ユーザーズ・マニュアル（日本モトローラ）
・MC68040 Designer's Handbook（Motorola USA）
・MC68881 ユーザーズ・マニュアル（電波新聞社）

DCACHE2. R

DCACHE2.Rは、ディスクキャッシュプログラムです。当初、Human68k ver 3に正式に対応していなかったので、X68030&Human68k ver 3になって添付されたfastioを使っていたのですが、RAMディスクのアクセス速度が半分になってしまうわ、「バースト転送」のメッセージはコケ脅しだわと散々でした。結局、X68000&Human68k ver 2の頃から愛用していたDCACHE2.Rをバージョンチェックを外して使うことに落ち着きました。

しかし、68040の動作で問題がないか心配で、内部を調べなければいけないかなあと思っていたら、作者のArimac氏*1が040turboの第一次配布に参加するとのこと。これはラッキーでした。

[BEEPs]

＊1
当時はX68030のオーナーでなかったにもかかわらず。

Arimac❖NIFTY-Serve：PAF03012

040turboとDCACHE2.RとBMPL.R

私が作ったソフトのうち68040で動かすうえで問題が出たのは、DCACHE 2.RとBMPL.Rぐらいです。というか、それくらいしか公開していませんし(笑)。公開していないソフトでもCで書いてあるのはまったく問題がないし、ハードウェアを直接操作しているソフトでも、あっさり動いてしまいました。

DCACHE2.Rでの問題点

DCACHE2.Rは、X68000では有名なディスクキャッシャソフト*2です。DCACHE2.Rの場合、68040で動かす際に問題になるのが、プログラムの自己書き換え*3でした。でも、常駐を開始するときだけのことなので、いったん常駐してしまえば問題はありませんが。

＊2
自分でいうなよ。

＊3
よくあるパターンです。

DCACHE2.Rでは2ヵ所で自己書き換えをする箇所があり、1つはデバイスドライバに割り込む部分の命令コードを作るところで、もう1つはキャッシュバッファを初期化するところです。

デバイスドライバに割り込む部分の命令コードというのは、

```
                .offset 0
DmdStrRtnEnt:   .ds.w   1   ;  move.l 5,    x
DmdReqHdrSav:   .ds.l   1   ;  リクエストヘッダアドレス退避アドレス
                .ds.w   1   ;  jmp          x
DmdOldStrRtn:   .ds.w   1   ;  本来のストラテジルーチンアドレス
DmdIntRtnEnt:   .ds.w   1   ;  jsr          x
DmdStrRtnAdr:   .ds.l   1   ;  割り込みルーチンアドレス
DmdStrRtnEnd:   equ     *   ;  align for 32bit
DmdMaxUnit:     .ds.b   1   ;  最大ユニットNo.
DmdUnitBase:    .ds.b   1   ;  内部ユニットNo.のベース値
DmdOldIntRtn:   .ds.l   1   ;  本来の割り込みルーチンアドレス
DmdDevHdrAdr:   .ds.l   1   ;  デバイスヘッダアドレス
DmdReqHdrAdr:   .ds.l   1   ;  リクエストヘッダアドレス
DmdSize         equ     *
```

となっています。

DCACHE2.Rは、実はR（リロケータブル）形式ではなく、X（非リロケータブル）形式にすれば自己書き換えをしなくてすむのですが、R形式にしたのは、単に私の趣味からです。

とはいっても、自己書き換えであれば、本来のストラテジルーチンへ飛ぶのはjmp命令でできますが、自己書き換えを行わないようにしようとすると、68000ではメモリ間接分岐ができないので、かなり冗長なコードを書く必要があります。68020以降ではメモリ間接分岐ができるのですが、そのためにはMPUの種別を判別しなければならず、やっぱり冗長なコードになります。

もう1つのキャッシュバッファの初期化は、キャッシュバッファを高位メモリにとらない場合に問題が出てきます。この場合、DCACHE2.Rは、常駐部のすぐ後からこのバッファを確保するのですが、常駐部の後というのは今まさに実行しているところで、そのまま初期化すると暴走してしまいます。

そこで、初期化するルーチン（最後に実行するところ）を、初期化時に内容の変わらないところ（バッファの中）へ移動しているのです。

どちらの問題も常駐開始時のことなので1ヵ所に（MPU）キャッシュをフラッシュするコードを入れることで解決してしまいました。

それが以下の部分です。

```
        Print   msg20(pc            ;'ドライブ'
        move.l  cache_drv(pc), d0
        bsr     mul_drv_print       ;' X:' ~ ' X:'
        Print   msg41(pc)           ;'をキャッシュ対象にします。¥n'
        cmp.b   #4,MpuKind.w        ;MPUは68040?
        bne     x_setup_500         ;<NO>
        move.l  #3,d1
```

```
                IOCS    _IOCS_AC            ;MPU CACHE CLEAR
x_setup_500:
                jmp     (a4)                ;常駐化処理(x_stay)へ
```

常駐化処理というのが、移動したキャッシュバッファ初期化ルーチンです。

BMPL.Rでの問題点

BMPL.Rとは、SX-WINDOWでMicrosoft-Windowsの壁紙（BMPファイル）を表示するプログラムです。768×512ドットモードで256色を同時表示するというのが特徴になっています。このプログラムにも、やはり自己書き換えを行っているところがありました。これもR（リロケータブル）形式ではなく、X（非リロケータブル）形式にすればすむところです。

もう1つの問題点は、BMPL.RにはMPUクロックに依存した部分があって、そこが障害となってうまく動作しなくなりました。本来、CPUクロックに依存したプログラムは書くべきではありませんが、BMPL.Rでは、1ラスターという、非常に短い時間（約22μsec）内に表示ページを切り替えなければならないため、適当なタイマがなく、命令の実行時間を計算することでタイミングをあわさざるを得なかったのです。

問題のルーチンがこれです。

```
x_hsync:
                Push_w d0                   ;8

                move.w front_cnt (pc), d0   ;12
x_hsync_100:    dbra.w d0,x_hsync_100       ;d0 * 10 + 14

                move.w #PAGE1,CRTC+VDC_R3   ;20

                move.w back_cnt (pc), d0    ;12
x_hsync_200:    dbra.w d0,x_hsync_200       ;d0 * 10 + 14

                Pop_w  d0                   ;8

                move.w #PAGE0,CRTC+VDC_R3   ;20
                rte    ;20
```

front_cntやback_cntのカウント値で時間を調整していることがわかるかと思います。行の右側に示した数字や計算式は、68000での命令の実行クロック数です。

X68000シリーズにはXVIやクロックアップしたマシンもあるため、CPUのスピードを計測し、それをもとにループカウント値を設定しているので、ある程度はCPUクロックに依存しないプログラムとなってはいるのですが、68000と68040では1つ1つの命令そのものの実行時間が違っているので、計算には多少違いが出てきます。

ところで、このBMPL.Rの機能を拡張しようと、*1ソースファイルがC言語で書いてあるモジュールを改造していくうちに、今まで表示できていたものが突然表示できなくなることがありました。原因を調べてみると、アドレスによって命令の実行速度が違うことがわかりました。68040は32ビットMPUであるため、命令長が4バイトの場合、アドレスによっては一度にアクセスできたり、二度に分けなければできないことがあるためでした。

ほかにも、スタティックカラムとかキャッシュラインとか、命令の実行速度に関係する要因はあるようで、68040の奥の深さを感じさせてくれます。*2

この問題を解決するには、fas*3というアセンブラで導入された、命令やデータを4バイト境界に整列させる.quadという命令を使えばよいのですが、リンカがこの命令に対応していなかったので、ちょっとした工夫が必要でした。

BMPL.Rは、ソースファイルがC言語で書かれているモジュールとアセンブラで書かれているモジュールから成り立っているのですが、C言語で書かれているほうがメインプログラムなので、これをアドレスの若いほうに配置し、アセンブラで書かれているほうはそれに続くアドレスに配置します。

C言語で書かれているプログラムがたまたま4バイト境界にあっていないアドレスで終わった場合、アセンブラで書かれているほうはリンカが対応していないので4バイト境界にあってないままリンクされます。これは、本来リンカが対応すべき問題かもしれませんが、とりあえず、C言語で書かれているプログラムの最後にasm命令で.quad命令を記述するという方法で解決しました。

C言語で書かれているプログラムの最後は、

```
asm ( "¥t.quad" ) ;
```

となっています。

68040も使ってみるまではいろいろ危惧する点もありましたが、今ではコピーバックモードを常用している状態で、意外になんとかなるもんだなあと思いました。もっとも、それは、BEEPs氏やじゃぎゅあ氏や、その他のX68000を愛する方々の努力があったからこそですが。

これを書いている最中にも、「少ないMPUキャッシュメモリを有効に使う

＊1
1678万色データの表示など。

＊2
調べてみればおもしろそうなのですが……。

＊3
Y.Nakamura氏とTNIのH.O氏が制作されたフリーソフト。

ため、DCACHE2.Rのキャッシュバッファに使用しているメモリをMPUキャッシュの対象から外している」という報告（NIFTY-Serve FSHARP 1会議室の書き込み）があったりします。というわけで、まだまだ040turboへのDCACHE2.Rの対応は現在進行形という感じです。

68040用デバッガ

68040では、X68000時代のデバッガDB.Xがいちおう使えましたが、スーパーバイザモードの動作が若干違うため、不便な思いをしていました。XC ver 2.1 NEW KITが出て、68030対応のDB.Xが登場しました。これでやっとまともなデバッガが使えると思ったら、こちらはもっと悲惨で、68040で起動するとハングアップしてしまいます。

ほとんどのソフトウェアは、少なくともキャッシュオフ状態なら68040モードでも動いたのに、デバッガだけはキャッシュオフでも動きません。これは手に負えそうもないかなとあきらめかけていたら、68040で使えるデバッガのパッチが公開されました。しかし、相当大変な作業だったようです。

［BEEPs］

なっち(湯浅夏樹)❖NIFTY-Serve：KHF03720

デバッガがない！

僕がX68030を注文したのは1993年3月4日(木)のこと。3月8日(月)にお金を払い込み、実際にX68030が届いたのは3月14日(日)のことでした。

それからしばらくは、X68000では動くが、X68030では動かないソフトのパッチ当てをやることも多かったのですが、そういうとき、いつも不便に思っていたことは「デバッガが使えない」ということでした。もちろん、[OPT.2]キーを押しながら立ち上げればROMデバッガが起動するので、これを利用することもできたのですが*1、ROMデバッガでは起動済みのプログラムしかデバッグできませんし、通常はRS-232Cでつないだ端末*2からデバッグすることになるので、ディスプレイやキーボードの切り替えが面倒です。*3 それに、通信ソフトからROMデバッガのコマンドを入力するため、HISTORY.Xによるキー入力のヒストリは使えないなど、どうも僕にはいまいち使いにくいものでした。

8月中旬、DB.X（ver2.10a）をX68030で動くように改造してみたりもしました*4が、とりあえず動作はするものの、バスエラー等が起こるたびにスタックポインタがなぜか少しずつずれていってしまうという代物になっていました。このスタックポインタの補正を完璧に行ったとしても、68030で拡張され

＊1
X680x0とROMデバッガの端末として使用するマシンのRS-232Cポートをクロスケーブルで接続し、端末側のマシンで通信ソフトを立ち上げておいてから[OPT.2]キーを押しながら起動させるとROMデバッガが利用できるようになります。X680x0でプログラム実行中になんらかのエラーが起こると制御がROMデバッガに移り、端末側の通信ソフトからROMデバッガのコマンドを入力できます。

＊2
僕の場合はACE-HD。

＊3
ACE-HDのキーボードからROMデバッガのコマンドを入力しないといけないのに、X68030のキーボードを叩きながら「なんで(ROMデバッガのコマンドを）入力できないんだ！」と思ったことも数知れません。

＊4
DB.Xの初期化ルーチン内でPrivilege Violationのベクタを書き換えないようにし、スタックフレームのアドレス計算を68030用に変更したら、とりあえずは動作するようになりました。

たレジスタや命令を扱うことはできないことを考えると、DB.XをX68030用にそれ以上改造する意欲は急速に薄れてしまいました。

9月中旬、シャープからXC NEW KITへのグレードアップの案内の手紙が届きました。X68030に対応したCコンパイラ、アセンブラ、デバッガを用意したということでした。以前のバージョンアップ料金から考えて、今回の料金も当然15,000円以下だろうと見積もっていたのですが、なんと2万円もしました。しかも、ver 2からのバージョンアップの場合でも、ver2.1からのグレードアップの場合でも同じ料金とのことでした。ちょっと頭にきたので、グレードアップの申し込みをするのは締め切りが迫ってからでもいいのではないかと思うようになりました。Cコンパイラは真里子さんのGCCを使うので必要ありませんし、アセンブラもH.OさんのFAS.Xがあるので、やはり、純正のAS.Xは必要ありませんでした。ほとんどデバッガのためだけに2万円も払うというのはちょっと抵抗がありました。

それからしばらくは、じゃぎゅあさんによるROMDB030.SYS*1を入手したこともあって、これを利用することでなんとかなっていました。

しかし、10月中旬頃、ふと「040turboと格闘するためには、68030対応のデバッガがあったほうがいいんじゃないだろうか」と思い、XC NEW KITの申し込みをしてしまいました。この結果、040turboが届く頃にはNEW KITも届いていて、68030対応のデバッガを利用できるようになっていたのです。

＊1
X68030のROMデバッガを、RS-232Cを通さずにX68030側で利用できるようにするツール。

040turboとの出会い

040turboのことを知ったのは、7月の中旬頃、SHUNAさんの運営するInside BBSで、SHUNAさんによって転載された040turboの紹介記事を読んだことがきっかけでした。その直後に、沖さんが「040turboの申し込みのメッセージを中継します」という書き込みをしてくれ、僕にとってはあこがれのCPUである68040を、うちのX68030で使うことができる！　ということで、さっそく申し込むことにしました。*2

このときには、僕はまだNIFTY-Serveには入会していませんでしたが、入会しておいたほうが後々なにかと便利だろうと思い、8月はじめにセゾンカードの入会を申し込み、カードが届いた9月下旬にオンラインサインアップでNIFTY-Serveに入会しました。

＊2
申し込みのときには、Inside BBSの沖さんやわかとのさんに大変お世話になりました。

今思えば、NEW KITの申し込み時期といい、NIFTY-Serveに入会した時期といい、どちらもベストタイミングだったように思えます。XC NEW KITを入手していたからこそ、XC NEW KITについていた68030対応のデ

バッガを68040で使えるように改造することができたわけですし、NIFTY-Serveに入会したおかげで040turboの情報をリアルタイムに入手できたことに加え、68040のジャンク品をPA５さんから安く譲っていただけたのです。

10月26日(火)、待望の040turboが届きました。はやる気持ちを抑え、とりあえず040turbo取扱説明書を見ながら必要なものがすべて入っていることを確認するだけにとどめました。こういうものは焦って取り付けても、(僕の場合は) うまくいかないことが多いのです。040turboの取り付けは、その次の土・日である10月30日、31日に行いましたが、このときには040turboのMPUソケットへの挿し込みの深さが足りなかった等でうまく動作させることができませんでした。*1

その後、BEEPsさんやPA５さんの助言を受け、11月２日、３日に再度チャレンジしたところ、なんとか無事動作するようになりました。*2

ところが、040turbo付属のフロッピーディスク内の040ERROR.LOGを見ると、不具合のなかに「030対応のデバッガが使えない」とありました。XC NEW KITが届いたときには「ようやく68030対応のデバッガが手に入った」と喜んでいたのに、

またデバッガが使えないのか……。

とがっかりしてしまいました。68040ではROMデバッガも動かないため*3、今度こそ本当に自分でなんとかする必要がありました。

＊１
このことからも、040turboが到着した日に取り付けに取り組んでいたら、大失敗していただろうことが想像できます。

＊２
このときには大変お騒がせしました。BEEPsさんをはじめ、FSHARP１のハードウェアの部屋のみなさん、ありがとうございました。

＊３
現在は、じゃぎゅあ氏によって、ROMデバッガを68040で動くようにするプログラム(ROMDB.X)が提供されています。

DB.Xとの格闘

11月７日(火)、ダメでもともとのつもりでXC NEW KITの68030対応のDB.Xを起動してみました。すると、X68030からの応答がなくなり、ハングアップ状態。やはりBEEPsさんの不具合報告のとおりでした。040turbo上で動作するデバッガがあれば、どこで止まるかわかるのでしょうが、DB.X自体が動かないうえ、ROMデバッガも使えない状態なので、とりあえずdis.x ver 2.06βを使用して逆アセンブルしてみることにしました。

```
dis -e -h -u -W a:¥bin¥db.x db.s
```

と入力して、68030対応DB.Xを逆アセンブルさせます。

ここでは、disに４つオプションを指定していますが、これらについて少し

説明しておきましょう。

〝-e〞は、ラベルファイル（db.lab）を出力させるためのオプションです。ラベルファイルを利用して、よりよい逆アセンブルリストを得るために指定しています。

〝-h〞は、プログラム領域をデータ領域と誤認識しても、その途中にある「$4E75(rts)」をきっかけに、できるだけプログラム領域として解析しようとしてくれるオプションです。未知のプログラムを逆アセンブルするときにはつけておいて損はありません。

〝-u〞は、未使用のA,F line trapを未定義命令と見なさないようにするオプションです。SX-WINDOW用のソフトを逆アセンブルするときには必須ですが、今回も68030専用のコプロセッサ命令*1を未定義命令と見なさないように指定しました。もっとも、dis.xは68030専用命令をサポートしていないので、このオプションを指定しても焼け石に水ではありますが。

〝-W〞は、絶対ショートアドレッシングを普通に出力するオプションです。AS.Xはver1.0の頃、絶対ショートアドレッシングを記述できないバグがあったため、dis.xはデフォルトでは絶対ショートアドレッシングをdc疑似命令で記述するようになっていました。しかし、現在ではAS.Xは絶対ショートアドレッシングをきちんと処理できるようになりましたし、フリーソフトウェアのHAS.XやFAS.Xでは、もともときちんと処理ができるので、これを指定して見やすい逆アセンブルリストを得られるようにしました。

dis.xは、X68000用のプログラムならフルオートで、かなり質の高い逆アセンブルリストを出力してくれるのですが、68030対応のDB.Xは68030専用の命令やレジスタを使用しており、dis.xはこれらの030専用命令があるブロックをデータ領域と見なしてしまい、逆アセンブルしてくれません。

そこで、68030の命令やレジスタに対応した逆アセンブラが必要となるのですが、ちょうどdisasmという、68020+68881用の逆アセンブラをX68000用に移植してありました*2ので、これを利用して、dis.xではデータ領域と見なされてしまう部分が実際にはどのような命令なのかを調べてみることにしました。

```
disasm -d a:¥bin¥db.x > db_disasm.s
```

disasmは、〝-d〞オプションを指定することで、実際のコードとニーモニックを同時に見ることができ、今回のような場合には便利です。

さて、disに〝-e〞オプションを指定してラベルファイルを出力させていま

＊1
未使用のF line trapになります。

＊2
FSHARP 1 のジャンクショップにアップロード済み。ただし、これはdis.xとは違い、単なる逆アセンブラなので、データ領域だろうがなんだろうが、全部命令だと見なして逆アセンブルしてしまいます。

したので、このラベルファイルを参照すれば、dis.xがどこをデータ領域と判断したかがわかります。

たとえば、ラベルファイルdb.labの一部を抜き出すと、

```
000128 P   colds
000140 DUF
0002CC DS
0002D0 DU
0002E0 P
```

となっています。

この書式の詳しい説明は、dis.x付属のマニュアルdis.docを参照してください。

ここで簡単に説明しておくと、左の数字はアドレスで、次の文字列の先頭の文字は、そのアドレスからの領域がプログラム領域と認識されている（P）か、データ領域として認識されている（D）かを表し、その次にもしラベルがあれば、これが記述されます。これを見ると、＄000140〜＄0002DFの部分がデータ領域と見なされていることがわかります。

db_disasm.sで、この部分がどうなっているかを見てみると、確かに$0001BCの「movec msp,a0」という命令あたりから68030専用命令が続いているようです。しかし、一見したところ、この68030専用命令の列は$000210のfmove.l命令を最後に終わっているように見えます。

```
0001BC 4E7A8803           movec   msp,a0
0001C0 2D4807D8           move.l  a0,(2008,a6)
0001C4 2D7C000155C60828   move.l  #0x155c6,(2088,a6)
0001CC 4E7A8804           movec   isp,a0
0001D0 2D4807DC           move.l  a0,(2012,a6)
0001d4 4E7A8000           movec   sfc,a0
0001D8 2D4807E0           move.l  a0,(2016,a6)
0001DC 4E7A8001           movec   dfc,a0
0001E0 2D4807E4           move.l  a0,(2020,a6)
0001E4 4E7A8002           movec   cacr,a0
0001E8 2D4807E8           move.l  a0,(2024,a6)
0001EC 4E7A8801           movec   vbr,a0
0001F0 2D4807F0           move.l  a0,(2032,a6)
0001F4 4E7A8802           movec   caar,a0
0001F8 2D4807EC           move.l  a0,(2028,a6)
0001FC 2D6E0172082C       move.l  (370,a6),(2092,a6)
000202 4A6E07D2           tst.w   (2002,a6)
000206 6730               beq.b   0x238
000208 F239D0FF00012BF6   fmovem.x  (0x12bf6).l,fp0-fp7
000210 F2399C00000129FA   move.l  (0x129fa).l,fpcr/fpsr/fpiar
000218 3D7CFFFF0834       move.W  #0xffff,(2100,a6)
```

このような68030専用命令の列は68000の命令とは見なすことができないので、dis.xがデータ領域と判断してしまうのもやむを得ないことといえます。そこで、仮にデータ領域は$0001BC～$000217の部分だけで、その他の部分はプログラム領域であると指定してみることにしました。また、$0002D0は命令の途中のアドレスを指しているので、この行をラベルファイルから削除しました。なお、ユーザーがラベルファイルを書き換える場合は、原則として小文字で書き換えなければならないのですが、この場合は$000140からの領域をプログラム領域と見なしてくれなくなってしまいます。

おそらく$000140から実行すると、$0001BCからのデータ領域まで実行してしまう*1ため、「本当のプログラム領域ならデータ領域に突入することはあり得ない。したがって、$000140からの領域はデータ領域であろう」と、dis.xが判断してしまうためではないかと思われます。

そこで、ラベルファイルの$000140の行については、dis.xがこの領域をもう一度解析することを抑制するために大文字で記述しました。*2

```
000128   P   colds
000140   P
0001BC   duf
000218   P
0002CC   P
0002E0   P
```

こうして、ラベルファイルの先ほどの部分に対応する部分を上のように変更してから、再度dis.xを実行してみました。*3

```
dis -e -g -h -u -W a:¥bin¥db.x db.s
```

すると、最初よりはいくぶんましな逆アセンブルリストを得ることができました。ただし、$000140～$0001BBの領域は無理やりプログラム領域に指定してしまったためか、本来ラベルをつけなければならない場所にラベルがついていない部分がありました。そこで、db_disasm.sと見比べながら、手作業でdb.sを修正していきました。*4

さて、こうして得られた逆アセンブルリストを目で追っていくのはかなり大変なので、ところどころに「$FF00」(DOSCALLの_EXIT)を埋め込んで、ハングアップ箇所を追い込んでいく方針をとりました。注意すべき点としては、今回の場合、dis.xでは正しくラベルが割り付けられた逆アセンブルリストは得られていませんので、「$FF00」の埋め込みはそれ以降のアドレスがずれな

＊1
本当はデータ領域ではなく、68030用の命令が並んでいるのですから、実行するのは当然のことですが。

＊2
大文字で記述しておけば、dis.xはそのアドレスからの領域は解析済みと見なします。

＊3
〝-g〞オプションを指定してラベルファイルを読み込ませています。

＊4
たとえば、アドレス$000162の「dbra d7,$0000015c」は本当は「dbra d7,L00015c」とし、ラベルL00015Cも本来あるべき位置に挿入しておかなければなりません。なお、手作業で修正しなくても、ラベルファイルに「00015c P」という行を付け加えてから逆アセンブルしなおせば、このラベルを得ることができるということには後で気がつきました。

いように行わないと、思わぬ誤動作の原因になってしまうということです。

具体的な作業としては、まずdb.sとdb_disasm.sとを見比べながら、db.sの適当な場所に「dc.w $FF00」を入れ、必要に応じてその付近の命令の削除や、nopやdcの埋め込み等をして、その後のアドレスがずれないように調整しました。そして、アセンブル、リンクして実行ファイルを作成し、それを実行してみて、コマンド待ちに戻ってくるか、それともハングアップしてしまうかを確認していきました。

はじめのうちは、律儀にDB.Xの実行開始アドレスから徐々に「$FF00」を埋め込む場所を後ろにずらしていったのですが、特に問題なくコマンド待ちに戻ってきました。後でよく考えてみれば、一番怪しいのは68030専用命令が並んでいる$0001BC〜$000217あたりですから、次にこの部分の前後に「$FF00」を埋め込んでみました。

すると、この部分の直前に「$FF00」を埋め込んだ場合はコマンド待ちに戻ってきましたが、直後に「$FF00」を埋め込んだ場合にはハングアップしてしまいました。

やはり、ここか。

と思い、「$FF00」を埋め込む場所を、db_disasm.sを参照しながらアドレス$0001BCからほぼ2命令ずつ[*1]ずらしていき、どこでハングアップするかを調べていきました。

ずらすこと約7回、

変だなぁ。なんでハングアップしないんだろう？

と不思議に思いはじめた頃、ようやくハングアップ箇所を突き止めることができました。それは、「movec caar,a0」でした。caarは68EC030には存在しますが、68040には存在しないレジスタなので、この部分でおそらくIllegal Instructionエラーが起こるのでしょう。しかし、この部分はまだDB.Xの初期化ルーチンの途中でしょうから、エラー処理ルーチンのベクタはきちんと設定されていない可能性が高く、そのきちんと設定されていないエラー処理ルーチンにジャンプしてしまうためにDB.Xはハングアップしてしまったのでしょう。

以前、68030と68040とを比較していたときに、

＊1
ハングアップ箇所は普通のmove.l命令のところではなく、movec命令のところだと予想していたので、movec命令の直後に「$FF00」を埋め込んでいきました。

へ～、68030にはあるのに68040にはないレジスタもあるんだ。

とcaarレジスタの存在は知っていたにもかかわらず、いざ、DB.Xが動かない原因を探るときには、このことには全然気がつかなかったということには苦笑せざるを得ません。

この「movec caar,a0」の部分だけ2つのnop命令に置き換えてからアセンブル、リンクして実行したところ、このDB.Xはとりあえず起動はするようになりました。しかし、qコマンドで終了しようとすると、エラーが起こって終了できません。

やはり、caarが使われているところを全部チェックする必要があるな。

と思い、db_disasm.s中のcaarレジスタをサーチして、そこに対応するdb.sのほうの命令をnopに変更していきました。変更箇所は全部で11ヵ所でした。

こうして、caarを使わないようにしたDB.Xを起動して、いろいろなコマンドを実行してみたところ、一応動いているようですし、qコマンドでちゃんと終了することもできました。

これで一件落着と思って、さっそく、このパッチ情報をNIFTY-ServeのFSHARP 1のジャンクショップにアップロードしました。

キャッシュクリア対応

SCD.XがハングアップするのもDB.Xと同じ原因だろうと思い、

```
disasm -d a:¥bin¥scd.x > scd_disasm.s
```

でscd_disasm.sを作成し、DB.Xのときと同様に、caarをサーチして、その部分をnopに変更するパッチを作成しました（これも手作業で11ヵ所)。これだけで、SCD.Xは起動するようになったので、ほっと一息つきながら動作試験として、鈴木国文さん作のFLOAT040.Xをステップ実行してみました。しかし、マウスでStepをクリックすると、するすると最後まで実行されてしまうではありませんか。

あれ？　今、何が起こったんだ？

と思ってしまいました。

そう、キャッシュのことを何も考えていなかったので、SCD.Xが設定した(つもりの) ブレークポイントはメモリには書き込まれたのでしょうが、命令キャッシュには反映されず、CPUはブレークポイントなんてないものと思って突っ走ってしまったようなのです。

あわててDB.Xでもステップ実行を試してみると、こちらも同じ症状が出ます。

アップロードするんじゃなかった……。

と思っても後の祭り。

とにかく早くキャッシュにも対応したパッチプログラムを作るしかありません。

もしかすると、68030モードでもキャッシュオン時にはステップ実行はできないんじゃないだろうか?

とも思って、68030モードでキャッシュオンにしてSCD.Xを実行してみたところ、ステップ実行は正常に行われているようでした。

これらの動作から考えて、ステップ実行ができないのは68030と68040とでキャッシュクリアの方法が違うことに原因があるのだろうと思い、040turboの取扱説明書を見てみると、ソフトウェアの説明のところに68030と68040ではキャッシュの制御方法が違うということが非常に詳しく書かれているではありませんか。

説明書をちゃんと読んでいる人なら、一番最初にチェックを入れるべき事柄ではないか。

と、我ながら恥ずかしくなりました。

DB.X内でキャッシュクリアをしているところをサーチしたところ、あるわあるわ、40ヵ所以上もあり、とても手作業で直す気にはなれませんでした。しかし、よくよく見てみると、キャッシュクリアをしている部分は全部以下のような10バイトの同じコードでした。[*1]

```
movea.l   #$808,a0     * 20 7c 00 00 08 08
movec     a0,cacr      * 4e 7b 80 02
```

＊1
これはSCD.Xでも同じでした。

そこで、このような10バイトの列を見つけたら、以下の10バイトに置換するプログラムを作って対処することにしました。*1

```
        cinva   ic/dc           ＊ f4 d8
        bra     LABEL           ＊ 60 06
        dc.b    08,08           ＊ 08 08
        movec   a0,cacr         ＊ 4e 7b 80 02
LABEL:
```

＊1
実際には置換されるのは先頭の4バイトのみです。

この置換プログラムでパッチを当てた後のDB.XやSCD.Xは、キャッシュオン時でも問題なくステップ実行ができることを確認し、急いでNIFTY-ServeのFSHARP1のジャンクショップに登録しなおしました。

しかし、このときには何も考えずにキャッシュクリアにはcinv命令を使っていました。cinv命令は、キャッシュ内容を強制的に無効化する命令であるため、キャッシュモードがライトスルーモードの場合にはこれでもいいのですが、コピーバックモードのときにはCPUが書き込んだはずのデータが（キャッシュには書き込まれていても）メモリには書き戻されずに消えてしまうといった現象が生じてしまいます。そのため、FSHARP1のハードウェアの部屋で、cinv命令よりもcpush命令のほうがよいと、中村ちゃぷにさんが教えてくださいました。

cpush命令は、キャッシュ内のデータをメモリに書き戻してからキャッシュ内容を無効化するため、コピーバックモードでも問題は起きません。また、せっかくIOCS-ACでキャッシュクリア機能が用意されているのですから、アプリケーション側からのキャッシュのコントロールは本来はIOCS-ACを使うのが望ましいと考えられます。

その頃は、

コピーバックモードがサポートされるのはまだまだ先の話だろう。

と思っていたので、のんびりかまえていたのですが、予想外に早くコピーバックモードがサポートされ、あわててIOCS-ACを使ってキャッシュクリアすることを考えてみました。しかし、僕の能力不足で結局断念してしまうことになったのです。

というのは、IOCS-ACでキャッシュをクリアするには、レジスタD1に3を入れてIOCS-ACをコールしなければなりません。ということは、コール前にはレジスタD0、D1を保存する必要があります。普通にコーディングすると、

```
movem.l   d0-d1,-(sp)
moveq.l   #3,d1
moveq.l   #$ac,d0
trap      #15
movem.l   (sp)+,d0-d1
```

となるのですが、これでは14バイトも消費してしまいます。しかし、パッチを当てるための領域は先ほど述べたように10バイトしかありません。D0レジスタが破壊されてもよいなら、

```
move.l    d1,-(sp)
moveq.l   #3,d1
moveq.l   #$ac,d0
trap      #15
move.l    (sp)+,d1
```

とすれば10バイトに収まりますが、D0レジスタが破壊されても大丈夫だという保証はありませんし、それについて40ヵ所以上*1にわたって調べ上げる気力もありません。

*1 キャッシュクリアしている部分は40ヵ所以上あります。

```
        cpusha  ic/dc       * f4 f8
        bra     LABEL       * 60 06
        dc.b    08,08       * 08 08
        movec   a0,cacr     * 4e 7b 80 02
LABEL:
```

結局、コピーバックモードへの対応としては、先ほどのパッチ中のcinv命令をcpush命令に置き換えただけのものを登録して現在に至っています。

僕自身はまだコピーバックモードはほとんど使用していないため、まだまだ問題があるかもしれませんが、問題が見つかれば、またプログラムを組むのにいいネタが見つかったと考えて対応していきたいと思っています。

謝辞

040turboという素晴らしいハード／ソフト（おまけに豪華取扱説明書）を開発してくださったBEEPsさん、68040を非常な安価で譲ってくださったPA5さん、また、有益な情報を書き込んでくださったFSHARP1のハードウェアの部屋のみなさん、040turboを知るきっかけとなったInside BBSのみなさんに感謝いたします。

MMUTM.X、ROMDB.X、そして040SYSpatch.sys

これらのプログラムは、68040と68030の違いからくるシステムプログラムの不具合を調整しようとするサポートプログラムです。もともと040SYSpatch.sysは040turboが動けばいいということで作成されましたので、いろいろと不備があり、これらのプログラムのサポートが必要でした。そのうち、これらの機能をどんどん内蔵して不備が取り除かれていったわけです。今や040SYSpatch.sysのことを一番よく知っているのは、じゃぎゅあ氏といっても過言ではないでしょう。

［BEEPs］

じゃぎゅあ❖NIFTY-Serve : NBH02724

040turboとスーパーバイザ保護

68040のようなMMUを内蔵しているMPUは、MMUの操作でメモリ保護が可能になっています。初期の頃の040SYSpatch.sysでは、OS管理下のメモリがユーザーエリアに設定されていました。それでもいちおう、process.xで表示されるHuman.sys領域はX68030のハードウェアによって保護されています。しかし、ユーザーモードで68040のMMUテーブルとパッチしたROM領域が変更できてしまうのはあまり気持ちがいいものではありません。

040turbo日記
1993年11月2日（火曜日）

怪しい予感がしたので、さっさと仕事をやめて秋葉原に行き、80486用冷却ファンとMC68030RC25の中古を購入した。帰宅すると、名古屋方面からの箱が到着していた……。

急いで部屋に行き、右タワー開放状態で040turbo装着を待つX68030を、定位置より引きずり出し、取り付け作業に入る。040turboの基板上のXC68040にシリコングリスを塗り冷却ファンを取り付けるのだが、80486用のために幅が狭い。ムリムリと金具の幅を拡げて、XC68040にキズがつかないようにかぶせるようにして取り付けた。

そこで、040SYSpatch.sysを変更せずにMMUテーブルだけを変更するMMUTM.Xを作ってみました。シェル起動後にMMUTM.Xを実行すると、68040のMMUによって、MMUテーブル、パッチROMの領域をスーパーバイザ保護するようにしました。さらに、お節介なことにHuman.sys領域もそうなるようにしました。MMUTM.Xは、Human.sysの最終アドレスが格納されているOSワークを$1C24からのロングワードで参照しています。

しかし、妙なことにソフトウェアリセット時にバスエラーが発生してしまいます。どうやらHuman.sysのブート時に一度だけMPUがユーザーモードになることがあるようです。それに、ソフトウェアリセット時にはMMUは生きたままになっています。この2つの条件が重なってバスエラーが発生するという、お節介が裏目に出てしまう間抜けな事態になってしまいました。

X680x0のRAMエリアは、リセット時、先頭の8Kバイトを除いてユーザーモードでの操作が可能になっています。しかし、MMUTM.Xを実行していると、生き残っているMMUのためにスーパーバイザ保護になっているところをユーザーモードで操作することになります。そのため、バスエラーが発生してしまうわけです。本来なら、リセット時にMMUを止めなければならないのですが、起動した後のプログラムではどうしようもありません。結局、お節介機能の、Human.sys領域のスーパーバイザ設定はやめ、X68030のハードウェアによる保護機能に頼ることにしました。

これでメモリ保護も完璧のように思えましたが、まだまだ甘かったのです。68040のキャッシュモードにはコピーバックモードがあります。このモードでは、リードのみならずライトまでもがキャッシュヒットし、データはキャッシュのみに書き込まれます。キャッシュからメモリへの書き込みはキャッシュプ

Voice of Users

冷却ファンの電源は、拡張スロットの+12Vから供給するように配線し、50MHzクロック信号線の引き出しを終え、X68030上のMC68EC030を外して、そのソケットに040turboを載せ、電源を入れた。

68040冷却ファンが動作しているのを確認し、プロンプトが出るまで待つ。

まずは030モードでconfig.sysに040SYSpatch.sysを書き加えた。040側にスイッチを切り替えリセット。「ぽ～ん」。動いたっ。

さっそくコンパイルやベンチマークをやってみるが、速い……。正確にX68030と比較するために030モードにもしてやってみた。

おっ？　げ……。

ハングった。

ッシュアクセスというメモリアクセスで行われますが、このファンクションが68030には存在しません。040turboでは、このファンクションを68030のスーパーバイザデータ空間にデコードしています。ユーザーモードでライトヒットしたデータも、スーパーバイザモードでメモリに書き込むことになるわけです。

次のリストを見てみましょう。

```
 1:  suba.l    a1,a1
 2:  IOCS      _B_SUPER          *  スーパーバイザモード
 3:  movea.l   d0,a1
 4:  move.b    $0000.w,d0        *  スーパーバイザで一度操作
 5:  IOCS      _B_SUPER          *  ユーザーモードに戻す
 6:  move.b    #0,$0000.w        *  ユーザーモードで操作
 7:  suba.l    a1,a1
 8:  IOCS      _B_SUPER          *  スーパーバイザモード
 9:  movea.l   d0,a1
10:  cpusha    dc                *  データキャッシュプッシュ
```

リストを見ただけで6行目でバスエラーが発生しそうなことがわかります。確かにキャッシュオフ時やライトスルーのようなメモリライト命令ですぐにメモリを操作するという状態では、X68030のスーパーバイザ保護機能によってバスエラーが発生するのですが、コピーバックモードでは4行目で$0000番地がアクセスされた後の6行目でライトヒットしてしまい、メモリ操作は行われません。そして、キャッシュのみに書き込まれたデータはユーザーモードで書き込んだにもかかわらず、10行目のキャッシュプッシュでスーパーバイザモードとして、メモリに書き込まれます。

X68030のハードウェアによる保護機能は、実際にメモリ操作が行われるときのファンクションコードをチェックしてバスエラーにしているので、このよ

Voice of Users

030モードで今まで動作していたソフトの一部がハングアップするようになってしまった。040モードで動かないのは正しい（?）が、030モードでハングアップするのは異常だ。

さっそくNIFTY-Serve FSHARP1　ハードウェアの部屋に報告して、ハングアップするソフトと、そのパターンを確実に再現する実験も行ってみた。

これを「030ぴーんち」と呼ぶ（ホントだよ）。

1993年11月5日（金曜日）

「030ぴーんち」の件で、040turboを作者のBEEPs氏に見てもらうことになったので発送。

うなケースではまったく意味をなしません。このようなことはめったにないと思いますが、ひじょーに気分が悪いといえます。これでは、下手をするとユーザーモードで容易にHuman.sys領域を操作（≒破壊）できてしまいますので、これは阻止しなければなりません。[*1]

では、どうしたらいいのでしょう？

これは、MMUでHuman.sys領域をスーパーバイザモードにすればよいだけのことで、MMUであればアドレス判定時に操作対象となるメモリのモードを検査するので、前出リストのようなケースでもしっかりバスエラーが発生します。このため、かつて一度失敗したお節介機能を復活させることになりました。

しかし、ただHuman.sys領域をスーパーバイザモードにするだけだと前に失敗したようにソフトウェアリセット時にバスエラーになってしまいますので、ブート時にMMUを止めるようにします。ここで、「勝手にMMUを止めて問題はないのですか？」といわれるかもしれませんが、電源投入時にはもともと止まっているものなので、まったく問題ありません。

では、次にどこでMMUを止めればいいのでしょう？　ブート時には、通常ROMから実行が始まりますが、040SYSpatch.sysではRAMに転送してパッチしたものを代用しているので、MMU停止コードを書き加えることは可能です。ブートプログラムの先頭で強引に空き領域にMMU停止コードを追加して、そこにジャンプさせてMMU停止を実行するようにするのもいいですが、あまりかっこがいいとはいえません。ブートプログラムを追うと、ちょうどよいことに、「MPUが68030か68040か？」を判別している部分があります。

これは、

＊1
これを悪用すると、040turboでしか動作しない変態ソフトが作れます。

Voice of Users

遅い……。

X68030が遅く感じる。この数日で68040 25MHzの速度に慣れてしまっていた。

1993年11月9日（火曜日）

チェックの結果、040turboに異常なし。問題はX68030本体にかかわることのようだ。

1993年11月13日（土曜日）

今度は、X68030と040turboを発送した。X68030の個体差による問題なのでセットで試すこととなった。

```
movec URP,A1
```

となっていて、68030に存在しない命令（68040専用）を実行して不当命令例外を発生させ、不当命令だった場合には68030と判別しているのです。

どうにかして、ここを使えないかと考えてみました。そうすれば、非常にスマートにMMU停止コードが収まります。MMUを止めるには、TCレジスタの第15ビットを0にしなければなりませんが、都合のよいことに、この命令にきた時点でD0レジスタが0になっています。ということは、

```
movec D0,TC
```

とすればよいのです。

TCレジスタは68030にも存在しますが、この場合にはpmove命令での操作になっています。これでMMUも止まりますし、68030では変更前と同様に不当命令例外が発生します。それに、この命令なら、変更前と同じ命令長でピッタリです。040SYSpatch.sysにパッチROMの該当部分を変更するように追加しました。[*1]

また、MMUTM.XではHuman.sysの最終アドレスを得るのにOSワークを参照しましたが、今回はHuman.sys内でスーパーバイザ領域設定ポート（以降、設定ポートと呼ぶ[*2]）を操作しているところにパッチをあててパッチプログラムにジャンプするようにし、パッチプログラム内でMMUを操作してHuman.sysの領域をスーパバイザ保護するようにしました。設定ポートを操作している部分は1ヵ所だけなので探しやすいし[*3]、さらに都合のよいことに

＊1
しかし、後で考えてみたら、パッチしたROMコード上で68040になっていることは明らかな状態ですから、こんなことをする必要はなかったのです。最新の040SYSpatch.sysでは、ソフトウェアリセット時にMMUを止めません。結局、空き領域にMMU制御コードを追加するという、かっこよくない方法をとってしまいました。

＊2
アドレス$E86001にあります。

＊3
アドレス$E86001（エリアセットレジスタ）をサーチするだけです。

Voice of Users

う、遅い。

15MHzに改造してあるX68000EXPERTだ。数日間は、これを再びメインで使うことになる。この日の東京は昼頃から小雨降る肌寒い1日だった。

1993年11月15日（月曜日）

BEEPs氏からのメールで「030ぴーんち」の原因が判明した。データバスに入っているバストランシーバのゲート信号のタイミングが、040turboを接続したことで不安定になり、間違ったデータをリードしている。これへの対策は、ゲート信号のピンに小容量のコンデンサを取り付ければよく、これで安定するようになったということだった。

は、設定ポートに出力する値は8Kバイトごとの値になっていて040SYSpatch.sysでMMUを制御しているページサイズと同じなので、設定ポートに書き込む値がそのまま使えるのです。

このパッチを行うソフトウェアは、デバイスドライバ型として登録され、実際に実行されるのはスーパーバイザ領域の設定時という少し変わった動作*1をするものとなりました。これを、HuSUPER.SYSとして発表しましたが、後に040SYSpatch.sys内に組み込んでもらうことにしました。

とりあえず、これで問題はなかったのですが、後でこの設定ポートが「バイトサイズ」であることを思い出したのです。

ということは、最大でも、

8K×256=2097152=＄200000=2Mバイト

という計算になり、先頭から2Mバイトしか設定できません。

ためしに、Human.sys領域内で2MバイトにRAMディスクを設定してみました。すると、やはりハングアップしてしまいました。どうやらHuSUPERのところでハングアップしているようです。調べてみると、Human.sys内で「スーパーバイザ領域が2Mバイトを超えた場合にはmoveq #$FF, D0してからD0の値を設定ポートに書き込んでいる」のです。設定ポートは8ビットですから困りません。最大値の2Mバイトになるだけです。

しかし、「moveq #$FF, D0」された値は32ビットのデータとして符号拡張されてしまい、実質的には#$FFFFFFFFとまったく同じことになってしまいます。悪いことに、HuSUPERではD0の値をページ数としてMMUテ

＊1
すべてのデバイスドライバが登録された後でないと、Human68kとしてスーパーバイザ保護すべき領域がわからないので、本プログラムの登録時はパッチをセットし、後ほどこのパッチを通るときにスーパーバイザ保護をします。

Voice of Users

1993年11月17日（水曜日）

X68030と040turboが帰宅し、現在に至る。はじめはたいした協力はできそうもないとか言っておきながら、今では68040の使い方を覚え、いくつかの謎めいたソフトができてしまった。68040的テクニックもいくつかわかった。

（文●じゃぎゅあ（大石健太郎） NIFTY-Serve : NBH02724）

ーブルの操作に使用していますから、これはとんでもない値です。メモリ領域の全域がスーパーバイザ保護されてしまいました。

これでは設定ポートに書き込むD0の値をそのままMMUのページ設定に使えないので、ページ数の計算は自前で行うように変更しました。ちなみに、これはX68030に始まったことではなく、X68000からある仕様でした。

040turboとROMデバッガ

ROMデバッガはオマケ的要素が強いソフトウェアです。なぜなら、ROMデバッガが動作しなくても、通常の使用ではまったく問題がないからです。しかし、なんとかしてROMデバッガを動かしたいと思いました。

X680x0では、[OPT.2] キーを押しながらリセットするとROMデバッガが起動するのですが、040turboでそれを行うとハングアップしてしまいます。これは、68040にはないレジスタを操作しているからで、これがなければ動くはずです。

そこで、040turboでROMデバッガを使用するうえでの問題点を整理してみました。*1

＊1
以降は、ROMデバッガを「ROMDB」と略します。

1．68040にはないレジスタ操作をなくさなければなりません。caarレジスタは存在せず、このままでは不当命令例外になってしまいます。そこで、これを相当する命令に変更しなければならないのですが、ここではnop命令に置き換えるだけでかまいません。
2．キャッシュクリアのコードを変更しなければなりません。
　変更しなくても動きますが、完全とはいえません。これは、cpush命令で置き換えることができます。
3．1.と2.の変更箇所は簡単に発見できますが、デバッガはROMにあるのですから変更することはできません。これでは無理です。

3.の問題をクリアするのは難しいことです。これを克服するには、デバッガをRAMに転送すればよいのです。しかし、RAMに転送した時点で、さらに次のような問題点が浮上しました。

4．ROMであるがゆえに、X形式の実行ファイルのようにリロケートの必要がなく、ROMDBのアドレスに依存した絶対アドレスを変更しているところがあります。これを転送したアドレスにあうように変更しなければなりません。これを行わなければ、処理途中でもとのROMにジャンプしてしまうからです。

絶対アドレス指定で使用する命令として、すぐに思いつくものを挙げてみましょう。

LEA, PEA, MP, JSR

などです。特に分岐命令などはそのままプログラムの実行にかかわってきます。

そこで、これらの命令のオペコードを頼りに探していくことにしました。オペコードの次のデータは必ずアドレス値で、ここがデバッガ格納アドレス範囲に入っていれば絶対アドレス指定なので変更の対象になります。C言語で検索プログラムを作成し、これを実行します。すると、pea命令では絶対アドレス指定が使われていませんでした。この検索プログラムで発見した絶対アドレス指定の箇所を変更して、RAM上で実行してみました。

動いた……。

なんとかデバッガっぽく動作するようになりました。

しかし、バスエラーが頻繁に発生してしまいます。やはり、ROMのほうにジャンプしてしまっているようです。

ROMDB内部の解析を進めると、アドレステーブルの存在に気づきました。この場合、4番地ごとにアドレス値が配置されているのですが、調べてみると、たまにそうではないものもあります。

困った。

モトローラの「PROGRAMMER'S REFERENCE MANUAL」を見ると、オペコードには一定の規則があることがわかります。命令の動作はオペコードの12~15ビットで分類されています。たとえば、〝0001〟であればバイト転送ですし、〝0111〟であればmoveq命令なのです。

今度は、このオペコードの上位4ビットを頼りに命令を分類して表示するプログラムを組むことにしました。当然、すでに判明している部分は無視することにしました。これで得られた結果をもとに、すべてDB.Xであたっていけばよいのですが、かなりの数があります。このためにバッチファイルを組んで、怪しいアドレスから少し前までを表示するようにし、すべての結果をリダイレクトしてファイル化し、確認することにしました。

アドレス値はテーブル構造になって格納されていることが多かったので、それほど確認する時間はかかりませんでした。こうして、すべての箇所を変更してRAM領域で再度実行しました。

動いたっ。

もうバスエラーも出ませんし、68030のときのようにデバッガからOSに復帰できるようにもなりました。ここまでくれば、後は使い勝手を向上させることだけです。コンソールからも使えるようにしたり、つねにメモリ内に常駐してオン・オフができるようにもしました。また、ROMDB専用領域の確保は、040SYSpatch.sysで行うようにオプション〝!〟を追加しました。

忘れてならないのはROMDBであるということで、そう簡単に破壊されては困ります。68040のMMUには、メモリにライトプロテクトを設定することも可能なので、この機能でデバッガ領域を保護することにしました。こうしておけば大丈夫です。MMUが止まってしまうと効果がありませんが、そういう事態になっているときにはROMDBですらまともに動作しません。

そして、このパッチプログラムをROMDB.Xとして発表しました。使い方はconfig.sysでデバイスドライバのように登録すると、ROMDBを専用領域に転送してリロケートします。使用する場合には、ROMDB.Xにオプションを与えてデバッガを起動します。後はROMDBと同じです。68040に対応させるというよりも、リロケートして動かすといったほうがいいような内容です。

最後にパッチしたアドレスの個数を列記しておきます。

caarレジスタ関連	9ヵ所
キャッシュクリア関連	44ヵ所
lea $XXXXXX	154ヵ所
jmp$XXXXXX	31ヵ所
jsr $XXXXXX	854ヵ所
その他	365ヵ所

総数……………………………………………1457ヵ所

かなりの数になりますが、後に1405ヵ所の変更が無意味になるのでした……。

68040とバスエラー

68040では、バスエラー例外は「アクセスフォールト」という例外に含まれますが、ベクタはバスエラーと同じで、ただ名前が変わっただけと思っていいでしょう。このバスエラーが困ったもので、スタックフレームが68030と異なるのです。それだけならいいのですが、スタックフレームにセーブされた内容には、「ライトバックバッファ」の内容や「プッシュデータ」が含まれており、これらのデータは本来ならばメモリに書き込まれていなければなりません。ROMDBでは、これらのデータを無視してしまいます。そして、Human68kでも無視しています。68040に対応していないので、当然といえば当然です。

もし、これが原因で重大な問題が発生するのであれば、そのうち誰かが対応してくれることでしょう。ただし、それ以前にバスエラーを発生するソフトウェア自体が重大な問題を持っているのですから、わざわざ対応する必要性を感じる人もいないと思います。「バスエラーが問題」ではなく、「バスエラーを起こすソフトウェアが問題」なのです。

パッチROM領域はRAMの中へ

040SYSpatch.sysでは、IOCS ROMを68040対応にするため、RAMに転送してパッチをあてています。このため、IOCSのアドレスがROMの場合と異なってしまいます。普通はちゃんとパッチをしているので特に困ることはありませんが、ソフトのなかにはたまにIOCSのアドレスに依存するものがあって動かない*1ことがあります。わざわざそういうソフトにパッチをあてるのも面倒ですし、パッチあてができない人もいるでしょう。これは、040SYSpatch.sysの問題ではないのですが、X68030との互換性が低いことになります。

互換性を得るには、パッチROMが元のアドレスに存在しなければなりません。そんなことは改造したり、ROMを焼いたりしなければできないことですが、68040ではその必要はありません。ソフトの変更だけで簡単に「パッチしたIOCSが元のアドレスに存在する」ようにすることが可能です。MMUのアドレス変換機能を用いて、パッチROM領域が元のアドレスになるようにアドレスを変換してしまえばいいのです。

一見すると、元のIOCS ROMとアドレスが重なってしまい混乱しそうですが、変換した値が有効なので、その点は大丈夫です。ただし、この操作で元のIOCS ROMはMPUからまったくアクセスできなくなってしまうので十分注意してください。

それでは困るという人のためにオプション〝0〟をつけました。これは、パッチROM領域を元のROMアドレスにするのとは逆の変換を行い、IOCS ROMをパッチROM領域に配置してアクセスできるようにするものです。このときにRAM領域に現れたIOCS ROMに書き込み操作を行うとバスエラーになるので注意しなければなりません。X680x0がそういう仕様なのです。

＊1
実際、そういうプログラムがありました。割り込みベクタの飛び先が＄FF0000以降、すなわちROM内でない場合、エラーとしていたのです。

哀・040turboとROMデバッガ

なぜ、哀しいのか。ROMデバッガは、ROMゆえにリロケートの必要性がなく、絶対アドレスで動作していて、68040で動かすにはRAMに転送してパッチをあてなければならないということは前に述べました。

しかし、パッチROM領域と同様に、パッチROMDBもアドレス変換すれば絶対アドレスへのパッチはまったく必要なくなります。ということは、ROMDB.Xでの絶対アドレスへのパッチはまったく無意味ということになります。

なんてこった……。

ね、哀しいでしょ。アドレス変換機能を使うことで、あまりにも簡単にROMDBが使えるようになってしまったのです。コードサイズも小さく、わざわざ別プログラムにするほどではないので、最終的にはROMDB.X相当の機能は040SYSpatch.sysに組み込み、オプション〝!〟でROMDBが使えるようにしました。通常のROMDBと同様にブート時に［OPT.2］キーを押していれば、ちゃんとROMDBが使用可能な状態で040SYSpatch.sysが起動します。*1

＊1
現在のバージョンではコンソールからの使用はできません。まったくROMDBと同機能です。

68040モード表示LED製作レポート

040turboを使う場合には、「68040だよLED」の配線が基板からコネクタを経由してびろ〜んと延びています。040turbo使用時には、その危険性を表すインジケータであるかのごとく赤く点灯します。これが結構邪魔なのです。LEDがなくても、68030と68040のどちらのモードであるか、スイッチの向きでわかるだけに邪魔なのです。

だからといって、このLEDを取ってしまうと、見かけ上X68030と同じになってしまい、040turboという感じがなくなり、かっこ悪くなってしまいます。また、LED用の穴を開けるというのは、きれいにできないとかえって見苦しい結果になってしまいます。

そうなると、右タワーの3個のLED（HD BUSY、TIMER、POWER）に細工することが考えられますが、3つともそれぞれ用途が決まっているLEDです。通常はこれらのLEDに細工をするのは不可能です。ましてや〝HD BUSY〟と〝TIMER〟を本来の用途以外に使うと非常に紛らわしくなり、なぜ点灯しているのかという理由もわからなくなります。これは〝POWER〟もそうです。

ただ、〝POWER〟LEDは、オフのときには赤で、オンのときは緑です。なぜ、ここだけ2色なのでしょうか。X68030を分解するとわかりますが、このLEDは内部に赤と緑の発色をする素子が入っているタイプです。このタイプのLEDは赤と緑を同時に点灯させたとき、橙色の発色をします。X68030では橙色状態は使用していませんので、この状態が68040モードとして使えます。ただし、そのままでは実現できませんので、少しハード工作をしなければなりません。

必要な部品は次の2つです。[*1]

トランジスタ 2SA1015（Y）（東芝製）……1
ダイオード　1S1588（東芝製）……………1

どちらも同等品や互換品でかまいません。それぞれ10円くらいで手に入ります。トランジスタには、ランク分けの存在する型番があり、専門店では同一型番でも区別して販売されています。2AS1015には、そのランクがあるのです。ここではLEDに供給する電流をスイッチングするだけなのでランクを問いませんが、心配であれば〝Y〟ランクと指定してください。実際に使用しているのは〝Y〟ランクです。

＊1
当然のことですが、ハンダや工作用工具もいります。

ダイオードは使用目的によっていくつかの分類がありますが、ここでは小信号スイッチング用を使用します。この回路では、電源オフ時に電流の逆流を防ぐために使用します。

最終的には図のような結線になるのですが、基板を使ったりはしません。すべては空中配線です。どのように組むかは作る人次第です。よく考えて作らないと、増設メモリや金具に接触してしまいます。ダイオードとトランジスタの足をうまく使えば、配線材の必要はありません。かといって、無理に足を曲げると折れたり、部品を傷めるので注意してください。

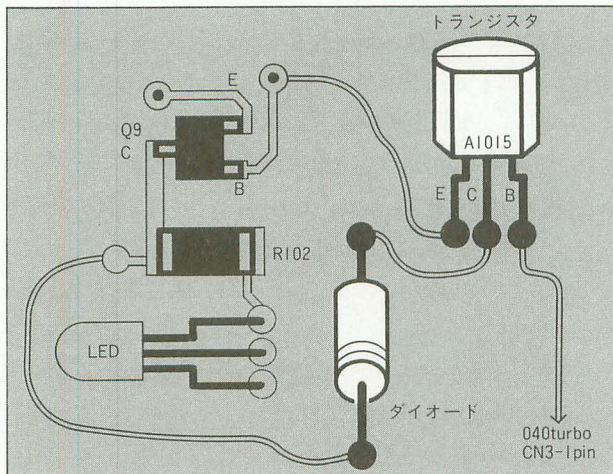



図　040turbo LED配線図(白い部分が新しく追加された部品)

配線材が必要な場合は040turbo付属の「68040だよLED」の線を使います。040turbo付属のLEDはいらなくなるので、線とのハンダを外して、ここから線材を切って配線用に使用します。これは5cmもあれば大丈夫でしょう。配線する際は、040turbo基板の「68040だよLED」のコネクタCN 3 - 1 pinに直接半田付けをせず、元と同じようにコネクタを通してつながるように配線しましょう。配線する際は基板上の部品をはがさないように注意して作業をしてください。増設メモリなどには接触しないように注意しましょう。

また、元からあったシールド板を使う場合にはフロントパネルの裏側にでも張り付けましょう。X68030基板のトランジスタ（Q 9）と抵抗（R102）は、〝POWER〟LEDのすぐ上にあります。

できあがったら動作確認です。68030モードで緑、68040モードで橙ならば正常です。

この回路はマンハッタンシェイプ型のX68030（CZ-500C）の場合です。compact型のX68030Compact（CZ-300C）の場合は基板がまったく別物な

ので、基板上の部品番号が異なっているでしょう。しかし、compact型の場合も、POWER LEDの配線をたどっていけば、同等の回路に到達するはずです。

なお、compact型には040turboが簡単には装着できないようなので、参考にはならないかもしれませんが。

さて、ここまでくると、勘のいい人は気づいていることでしょう。モード切り替えスイッチはどうしたのかということに。これはぶら下がったままなんですよね……。あー、かっこ悪い。

040turboとUNIXのアヤシイ関係!?

NetBSD

いろんなパッチプログラムによって、040turboでHuman68kはなんとか使えるようにはしたものの、「やっぱ、040にHumanじゃもったいないよな」という感じは拭えません。これでは、速い68000にすぎず[*1] 68040が泣きます。

68040は、かのNeXTにも搭載されていたプロセッサです。MMUも載ってるし、ワークステーション並のパワーもある。UNIXが載るのに十分なハードウェア[*2] は整ったといえるでしょう。これでUNIXが動いたら完璧です。X68000のキャッチコピー「パーソナルワークステーション」がハッタリでなくなるのに、と思っていましたら、出てきました。

沖さんが、NetBSD[*3]を移植しているというのです。なんとタイミングのいいことでしょう。「040turbo+NetBSDでパーソナルワークステーションを実現せよ」との神の意思を感じとり、ほとんど押売りに近い状態で040turboを渡して68040への対応を迫ったのでした。

［BEEPs］

＊1
80386、486といったプロセッサを、速い8086としてしか使えないMS-DOSを笑ったものですが、同じ立場になってしまったわけです。

＊2
ちょっとハードディスクアクセスが遅いけど。

＊3
UNIXの1つ。後述。

沖＠沖❖NIFTY-Serve：GGC02412

X680x0 とUNIXを結ぶ線

UNIXというのは、数あるオペレーティングシステム(Operating System；OS）のうちの1つです。オペレーティングシステムとは、ハードウェアとアプリケーションソフトウェアとの間で機能するプログラムで、ハードウェアの差異を吸収したり、アプリケーション間の共通機能を提供したりするプログラムです。MS-DOSやOS-9も、皆さんよくご存じのHuman68kも、オペレーティングシステムの1つです。C言語の本でよく解説されるように、UNIXはそのほとんどの部分をC言語で記述されています。

このUNIXは、ワークステーションという、パソコンより一段高級なコンピュータ上のオペレーティングシステムとして使われていて、最近ではソフトウェア研究開発の分野でほぼ業界標準の地位にあるマルチタスク・マルチユーザーのオペレーティングシステムです。また、UNIXは、コンピュータ同士でのデータのやりとりを行うためのネットワーク機能も有している強力なOSです。

一見、このUNIXとX680x0は関係ないように思われるかもしれませんが、実はUNIX上の資産の一部はX680x0で活用されています。たとえば、「X68k Programming Series」(ソフトバンク刊) でも有名な、GNU C Compiler (GCC) は、もともとUNIX上で動いていたものが、Human68kでも動くように手直しされ、大幅に改良されたものです。GCCのみでなく、GNUと名のつくものはMakeもEmacsもすべて、もとはUNIX上で動いていたものです。

GCCのほかにも、UNIX上で動いていたものがHuman68k上で動くように手直しされ、活用されているソフトウエアはたくさんあります。業務利用も可能な組版システムであるTeXがHuman68k上で動作していますが、これもUNIX上で動作しているものを移植*1したものです。

また、もともとUNIXに備わっている機能をHuman68k上で実現、もしくは疑似的に再現するために作られたソフトウェアも存在します。たとえば、拙作lndrvは、現在ではすでに標準的なシンボリックリンクの機能をHuman68k上で実現するものです。シンボリックリンクは、4.2BSD UNIXで新たに加わった機能です。また、やはり、拙作execdは、UNIXでの実行属性の概念をHuman68k上でも再現し、「シェルスクリプト」と呼ばれる一種のバッチファイルを実行できるようにするものです。さらに、ITA TOOLBOXと呼ばれる一群のソフトウェアは、UNIX上に標準的に用意されているソフトウェアと同等の機能をHuman68k上で実現しています。

*1
ただし、preview.x、print.xはHuman68k上のオリジナルソフトウェアです。

これらのソフトウェアを使えば、Human68kを動かしているのにもかかわらず、UNIX上とほとんど同じ操作を実現することもできますし、実際、そうしているX680x0ユーザーもいます。普段command.xを利用しているユーザーが「Human68kだとは思わなかった」というほどです。

また、X680x0上で使われている有名なエディタにMicroEmacsがありますが、このMicroEmacsのもとになっているEmacsもUNIX上で動くものです。

このように、X680x0とUNIXとは見えない糸で結ばれているのです！

しかし、これらのソフトウエアがUNIX上で動いているのと寸分違わずにHuman68k上で動いているかといえば、答えはノーです。

たとえば、本来UNIXで動作するNemacsは、「シェルモード」といって、エディタ上でシェルプログラムを並行して動作させることができます。しかし、このシェルモードの機能は、現在のHuman68kでは実現することができません。なぜかといえば、現在のHuman68kは「任意の複数のプログラムを、見かけ上同時に動かす」という機能が備わっていないからです。

もちろん、なかにはその性質からHuman68kに移植できずにいるUNIX上

のプログラムもたくさんあります。たとえば、UNIX上で動作するウィンドウシステムの業界標準であり、また、フリーソフトウェアでもあるX Window Systemがそれにあたります。

OS-9/X680x0上に移植されていることから、うすうす感づいておられる方もいらっしゃると思いますが、このX Window Systemは、オペレーティングシステム上で複数のプログラムが同時に動いていることを前提として作られているのです。*1

なぜX680x0でUNIXは動かない？

しかし、このようなオペレーティングシステムがあったら、X680x0に移植されてもおかしくないような気がしませんか？ 実際7年前の「Oh!X」誌*2には「シャープが、UNIXもしくはUNIXライクなOSをX68000用に提供することを考えている」という文章が掲載されています。*3

しかし、実際にはUNIXはX680x0には移植されませんでした。

最近、「PC UNIX」と称してフリーソフトウェアのUNIXクローンが何種類かPC/AT互換機ユーザーなどの間に広まっていますが、このUNIXクローンですらX680x0で動くことはないのです。

なぜでしょうか？

これには大きな理由があるのです。68000の10MHzでは動作自体が遅いといった理由からではなく、そもそも動作させること自体が不可能なのです。比較的最近のMPU（CPU）である68030、68040や80386、80486など、さらにはワークステーションで使用されている一連のRISC（Reduced Instruction Set Computer；縮小命令セットコンピュータ）などには、たいていメモリ保護や仮想記憶のためのメモリ管理機構が内蔵されています。

*1
X Window Systemは、米マサチューセッツ工科大学の商標です。

*2
1987年2月号連載コラム「Again Watch」。

*3
今から考えると、OS-9がこれにあたるようですが……。

C O L U M N

X680x0のメモリ保護機能

X680x0は、ハードウェアからスーパーバイザエリア指定を行うことができ、0番地から8Kバイト単位の連続領域を指定することで、メモリの先頭2Mバイト分だけをユーザーモードで書き込みができないようにすることができます。仮想記憶機能のあるMPUでは、この機能を拡張したような感じで、区分けされたメモリ領域に対して、任意に読み書きの禁止／許可を設定できるようになっています。メモリ保護機能があると、あるプロセスがOS自身や他のプロセスのメモリを破壊できないようにすることができます。

Human68kでは、各プログラムで連続したメモリ空間を使うことになっていますので、常駐プログラムを解除したりすると、メモリ空間の途中に分断された空き領域ができてしまいます。残念ながら、あるプログラムが連続した大きなメモリ領域を必要としているときには、この分断された空き領域を使うことはできません。しかし、もし仮想記憶機能があったならば、この空き領域を利用することが可能なのです。

どういうことかというと、仮想記憶を使うと、プログラムから見たとき、ハードウェアとして用意されているメモリ番地を、別の番地になるように変更することが可能なのです。

また、X680x0では実装できるメモリ容量が最大12Mバイトとなっています。通常、Human68kのプロセスはつねにメモリ上に存在しますから、子プロセスを次々と動作させていくと合計12Mバイトを超えてしまいます。しかし、実際には、この12Mバイトを超えたメモリを使うことができません。そのため、「メモリが不足しています！」「Memory exhausted.」といったメッセージに遭遇することになります。SX-WINDOWであれば、Xアイコンが赤くなるといった状態になり、どちらの場合もやろうと思ったことを実行することができません。

仮想記憶（virtual memory）を利用すれば、その名のとおり、実際には存在しない記憶領域があたかもあるかのように見せることができます。しかし、仮想記憶機能は、残念ながら、X680x0のMPUである68000や68EC030には用意されていません。これらが大きな要因となって、X680x0ではUNIXが動かないのです。

040turboで動き出すUNIX

X680x0ではUNIXは動かないと書きましたが、よく考えてみましょう。

そういえば、X68030のMPU68EC030はソケットに装着されているので、メモリ管理ユニット（Memory Management Unit；MMU）を内蔵した68030と差し替えることができるではありませんか。そして、同じくMMUを内蔵している68040を、X68030のMPUとして動作させる040turboプロジェクトもあるではないですか。

これらの、半田付けも不要の簡単な改造でX68030にもUNIXが動く条件が整うのです。もし少しでもUNIXに近づきたいとすれば、そして仮にもUNIXが動くのであれば、これらの改造をすることに何のためらいがあるでしょうか。

しかし、マシンを改造してもX68030用のUNIX自身がぽんと勝手に生まれ

てくるわけではありません。

そこで、「ないものは作ってしまえ」とばかりに、私はフリーソフトウェアとして配布されているUNIXクローンの入手および移植を行おうと考えました。C言語で書かれていれば、ハードウェアに依存している部分をX68030用に書き直すだけで、UNIXクローンが動き出すだろうと考えたわけです。

当時入手可能なUNIXクローンはいずれもPC/AT互換機用のもので、候補はいくつかありました。そのなかでも、他機種への移植が考慮されつつあったNetBSDにターゲットを絞ることにしました。*1

この移植を思い立った当時は、040turbo自体がまだ完成していなかったため、68030で動作するNetBSDを作ることにしました。

＊1
NetBSDは、実際にAmigaなどに移植され、動作しています。

NetBSDを移植する PART 1

移植を始めるにあたり、まず、NetBSDの移植を行うのに必要なものを整理してみました。

1. オリジナルのソースプログラム
2. ソースプログラムをコンパイルできる環境
3. コンパイルしたNetBSDを動かすハードウェア
4. NetBSD上で動くプログラムの入ったディスク
5. NetBSDを立ち上げるためのプログラム

1.のオリジナルのソースプログラムはNIFTY-Serveなどに登録されていましたので、それを入手することにしました。

問題は、2.のソースプログラムをコンパイルする環境です。最初に困ってしまったのは、「どうやってコンパイルすればいいのかわからない」というものでした。というのも、私はNetBSDをコンパイルしたことがないどころか、動いているところさえ見たことがなかったのですから、しかたありません。

NetBSDは､仮想記憶機能を使う68030/68040専用のプログラムになります。当時は、まだ68030の命令を使うことのできるコンパイラやアセンブラがありませんでしたから、コンパイラやアセンブラを独自に用意することにしました。

次に、NetBSDのコンパイル手順を調べて、必要なプログラムをHuman68k上で動くようにすることにしました。

そこで、用意した主なプログラムは、

・GNU C Compiler ver1.42（Cコンパイラ）
・GNU Assembler ver1.38（アセンブラ）
・GNU binutils-2.2　ld（リンカ）
・pmake（専用makeプログラム）
・config（専用のコンパイル環境設定プログラム）

の5つでした。

移植したコンパイラなどは、実際に移植したものが動いてくれるまで動作確認できないのが不安だったのですが、それはそれで動かなかったときにどうするか考えようということにしました。

いろいろ苦労した結果、なんとかHuman68k上でNetBSDをコンパイルすることに成功しました。

3.のNetBSDを動かすハードウエアですが、これは当然X68030です。もちろん、先ほども書いたように、買ってきたままのX68030ではNetBSDは動かないわけで、MPUを68030に差し替えたり、040turboを装着したりする必要があります。

4.は大きな問題でしたが、X68030への移植作業中にNetBSDのAmiga版が公開され、そのAmiga版のディスクイメージ[*1]を入手できたために無事解決することができました。[*2]

5.の立ち上げ用プログラムは、コンパイルできた実行ファイルを実際に動かすというものです。普通のHuman68k用のプログラムであれば、コマンド名を入力したり、SX-WINDOW上でアイコンをダブルクリックすることでプログラムを動かすことができますが、NetBSDはHuman68kやOS-9などのようにOSそのものですから、そういうわけにはいきません。本来はIPLから直接起動、といきたいところですが、面倒なのでHuman68kからNetBSDを起動できるようなプログラムを作ることにしました。

と、まあ、こんな具合にゆっくりと移植作業は始まりました。

円滑なマルチタスク処理を行うため、NetBSDではハードウェアの制御にIOCSを使わず、直接各LSIを制御することにしました。そのためにX68000のハードウェア関連書籍とにらめっこしながら処理部分を作っていきました。

＊1
NetBSDで使っているHDの中身をそのまま吸い出したファイル。

＊2
最初は、このディスクイメージを使えるかどうかわからなくて苦労しました。

NetBSDを移植する PART 2

移植作業を始めて４ヵ月、68030にMPUを差し替えたX68030上でNetBSDがどうにか動き出した頃、040turboも第一次配布を行うところまできていました。私は、残念ながら、この第一次配布には参加できませんでしたが、このときに040turboでNetBSDが動かないかという話があり、「040turboの030での動作が安定したら、やってみたい」といっておきました。

そのうち、第二次配布の募集が始まったのですが、このときにも個人的な事情で参加を断念しました。こんなことで040turbo上でNetBSDが動く日はくるのでしょうか？

ところが、ある日、040turboの作者のBEEPs氏から「040turboをお貸ししますから、やってくださいよぉ」と、半ば強制（？）的に試作版の040turbo[*1]を渡されてしまいました。

＊１
040turbo製作編でも紹介してありますが、基板の向きが違っているというヤツです。おかげで、現在もX68030についている040turboはマザーボードぷらぷら状態です。

こうなったらやるしかない！

C O L U M N

NetBSDでの030/040判定方法

NetBSDでは、040SYSpatch.sysで行っているのとは別の方法で、MPUが68030か68040かを判定しています。68030と68040とではCACRレジスタの持っている機能が違っていますので、これを利用しています。すなわち、68030のCACRレジスタがbit０からbit13までを使用しているのに対して、68040のCACRレジスタではbit15とbit31の２つだけを使用しています。この68040で使われている２つのビットは、68030のCACRではつねに０となっています。このため、bit15とbit31を１にした値をCACRに書き込み、次にCACRを読み出して、このビットが１になっていれば68040、０であれば68030であると、判定することができます。

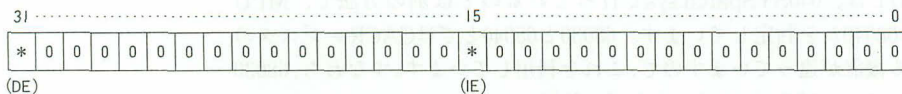
もちろん、このビットはキャッシュ制御に関する重要な意味を持っていますので、判定に使った後は元の値に戻しておくべきでしょう。

68030のCACRレジスタ（＊は０か１を設定可能）

040turboは、MPUが68030と異なる点以外には違いはありませんから、ハードウェアの違いで苦しむこともないはずです。MPUの違いのうち、ソフトウェアから見た最も大きな違いは、MMUです。Human68kとは違って、NetBSDではMMUをフル活用していますから、変更箇所も多くなります。さらに、キャッシュ関連も68030と68040とでは違っていますので、変更が必要です。

Human68kの040turbo対応がそうであったように、OS本体を040turboに対応させさえすれば、ほとんどのアプリケーションは個別に040turbo対応を行わずにそのまま使うことができます。Human68kでは、040SYSpatch.sysがMPUの種類を判別し、自動的に040turbo対応を行うようになっていますが、NetBSDではOS本体自身のソースプログラムがあるわけですから、各部分に040turbo対応コードを埋め込んでいきます。

このようにしてNetBSDの68040への対応が進んでいったわけですが、さらに問題がありました。浮動小数点演算関連です。X68030では、浮動小数点演算コプロセッサ（Floating Point Co-Processor；FPCP）として、68882を後から追加できるようになっていますが、68040には68882の機能縮小版がは

68040のCACRレジスタ（＊は0か1を設定可能）

リスト1：030/040　MPU判定ルーチンのリスト

```
/*  MPU check (-030 or 040) code
 *  return: d5.1 0...not040 1...040
 */
MPUcheck040:
        movc    cacr,a0
        movl    #0x80008000,d5
        movc    d5,cacr
        movc    cacr,d5
        movc    a0,cacr
        cmpil   #0x80008000,d5
        beq     MPUcheck_1
        moveq   #0,d5
        rts
MPUcheck_1:
```

じめから内蔵されています。問題となったのは、この内蔵FPCPで削られた機能の実現です。

最初に作った040turbo対応版NetBSDは、削られたFPCPの機能を使用すると、OS本体が異常終了してしまうというものでした。それでもシングルユーザーモードでの起動には成功し、lsなどのプログラムも動作しました。しかし、浮動小数点演算を要するプログラムの多くは動きませんでした。

たとえば、ディスクの容量および使用量を表示するプログラムdfは、残量が全体量の何%であるかを表示します。この割合を計算するときに浮動小数点演算命令を利用するためにdfは動作しませんでした。

なんとかしなければと思い、040turbo専用のHuman68kの浮動小数点演算ドライバからコードを拝借して浮動小数点演算機能をフル実装できないかと考えました。そして、中村ちゃぷに氏のPFLOAT.Xをもとにどうにか組み込んではみたものの、症状は変わらず、結果は失敗に終わりました。

そんなわけで浮動小数点演算をあきらめかけていたところに、救世主が現れました。内蔵FPCPで削られた機能をソフトウェアでエミュレーションするFPSP（Floating Point Software Package）のオブジェクトが、NetBSD

```
        moveq   #1,d5
        rts
```

リスト2（Motorola syntax）

```
# MPU check (-030 or 040) code
# return: d5.1 0...not040 1...040
#
MPUcheck040:
        movec   cacr,a0
        move.l  #$80008000,d5
        movec   d5,cacr
        movec   cacr,d5
        movec   a0,cacr
        cmpi.l  #$80008000,d5
        beq     MPUcheck_1
        moveq   #0,d5
        rts
MPUcheck_1:
        moveq   #1,d5
        rts
```

/Amiga用として公開されたのです。このオブジェクトを利用することができれば、040turboでの浮動小数点演算処理が完璧になります。

さっそくFPSPを組み込もうとしたのですが、どうもHuman68k上でのクロスコンパイル環境のオブジェクトとはフォーマットが違うらしく、うまくカーネル*1を作成することができません。030モードでの動作が安定してきていたこともあって、思いきってコンパイルをNetBSD上で行うことにしました。ファイルをNetBSD上に転送し、コンパイルしてみると、当然ですが、本来のコンパイル環境であるためか、すんなりとリンクできました。後はこれを動かすだけです。タイミングよく、中村ちゃぷに氏がNetBSDのIPL起動プログラムを発表されましたので、これを利用して起動してみました。

無事起動したので、まずは、以前動作しなかったdfを実行してみます。見事に動作し、ディスク残量も表示されました！

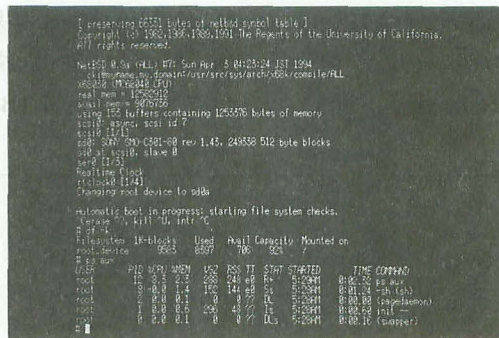



写真１　df実行画面

いよいよ040モードで、NetBSDのマルチユーザーモードでの起動を試してみます。オープニングメッセージ、ディスクの接続状況、ディスクの内容チェック、次々と起動時の動作が行われます。しかし、ネットワーク関係の設定の途中でダンマリ状態になってしまいました。原因を調査する必要がありそうです。

と、ここまでが1994年３月現在の状況です。まだNetBSDは040turboに完全対応したとはいえませんが、これからの対応作業で無事動作するだろうという感触を得ることはできました。

＊１
ここでは、OS本体を意味します。

040turboとNetBSDのこれから

NetBSDでサポートしているハードウェア資源は、96桁×32行の日本語表示が可能なディスプレイやキーボード、SCSI HDD/MO/CD-ROM*1、RS-232C*2となっています。これも、将来はグラフィック機能をサポートするなど、さらなる機能拡張を進めていければと思っています。もちろん、040turbo完全対応も目標の1つです。

NetBSDはソースプログラムが公開されているOSですから、誰でも内部の仕組みを知ることができますし、改良や機能追加も可能です。ユーザーとしての立場だけでなく、こうして開発者になることも夢物語ではありません。Human68kとは方向性の異なる、このNetBSDがより多くの方に愛されるものになれば、そして、040turboとともに末永く使っていただければ、と願っています。

＊1
現時点では、MO/CD-ROMの使用中にメディアの交換ができません。

＊2
現時点では、ハードフロー制御をサポートしていません。

付録

1. 040turbo取扱説明書
2. 040turboアプリケーションソフトウェア動作状況

第1章
040turboの概要

040turboは、X68030のMPUである68030にかわり、68040を使用することを目的に開発したフリーハードウェアです。X68030から68030を取り外して空いたソケットに接続するドータボードという構成をとっています。ただし、MPUを載せ換えて動作速度を速くする、いわゆるアクセラレータではありません。ソフトウェアのパッチも必要ですし、不具合の出るプログラムもあります。

ソフトウェアの互換性を考えれば、無理に68040を使うよりも、68030をベースにしてクロックを上げたり、外部キャッシュを積んだりしたほうが妥当でしょう。しかし、あえて68040にこだわっているのは040turboが68040を動かすこと、それ自体を第一の目的としているからです。この点を、まず理解してください。

040turboに搭載した68040自体に、ハードウェア面、ソフトウェア面ともに68030と完全互換でない部分があるので、すべてのプログラムについて完全に問題を解消することは不可能ですが、ハードウェアの相違に関しては040turbo基板上の変換回路で、ソフトウェアの相違に関してはパッチプログラムでかなり対処できています。

パソコン通信を通して行った2回の配布以来、多くの人の手によって68040対応のプログラムが〝フリーソフトウェア〟として公開され、68030との違いによる問題点は克服されつつあります。現在では、68040で動いているのを意識することはほとんどないくらいのレベルになっているといってよいでしょう。

簡単な速度比較の結果を表1.1に示します。040turboは、通常のキャッシュオン時の動作では、68030の約2倍の性能を発揮しています。もっとも、まだまだ手が回りきれていない部分も多いのは事実です。特に音楽関係のプログラムへの対応が進んでいません。X68000からX68030へ移行したときでさえ、いろいろな問題がありました。ましてや、040turboは個人的にコツコツと製作してきたものであり、今、やっとできたばかりというところです。どの部分がどう違っているのか、問題点は何かなどについては第4章、第5章でできるだけ詳しく説明したつもりですので、ハードウェア、ソフトウェアについて、みなさんの手で積極的に改良を加えていってください。また、不明な点や不具合箇所を発見したときは報告をお願いします。可能なかぎり対処するつもりですが、〝フリーハードウェア〟と〝フリーソフトウェア〟ですから、保証はできません。その点をご理解ください。不具合報告の書き方についてはAPPENDIX Aを参照してください。

なお、040turbo基板上にはX68030から外した68030を搭載することが可能（搭載しなくてもかまいません）で、スイッチにより動作MPUを切り替えることができます。

68030モードの場合は、基本的にはX68030の本来の動作と変わらないはずですが、040turbo基板の実装による信号線長や負荷の増大によって不具合が出る場合があります。68030モードでエラーが頻発する場合は、X68030のマザーボード上での対処が必要になります。詳しくはAPPENDIX Bを参照してください。

プログラム	68030	68040	備考
dhry.x	6578.9	15625.0	Dhrystone Benchmark, ver2.1 (Dhrystones/sec)
whet.x	657.89	1449.28	Whetstone Benchmark, ver1.0 (KWhets/sec)
pv.x	5.48	16.42	X68000 10MHz比
キャンバス.x	約17秒	約8秒	SX-WINDOW上で「草原.JPG」が表示されるまでの時間

＊キャッシュオン（68040はライトスルーモード）で実行

表1.1　040turboと68030の速度比較

1.1　040turboとX68030の接続について

040turboでは、68040を動かすことが目的なので、極力シンプルなハードウェア構成をとっています。このため、X68030への取り付けは68030のすべての信号線を乗っ取ることのできるMPUソケットへの挿し込みという形をとっており、拡張スロットや、コプロセッサの取り付けといった、簡単な作業ではすみません。本体の分解を含めた大仕事になりますので、十分に作業内容を理解したうえで行ってください。また、マザーボードを覆うシールドを外してしまうことになるので、テレビやラジオにノイズが出たりするようになると思います。

実際の作業については第2章で説明していきますが、ここでは、040turboとX68030の間で接続する3種類の信号線について先に説明しておきます。

1. 68030のMPUソケットを介した信号
2. 50MHzのクロック信号
3. MPU切り替えスイッチと表示LEDの信号

実際の取り付け作業は、簡単にいってしまえば、これらを040turboとX68030の間で結ぶ作業ですが、何のためにこれらの接続を行うのか理解したうえで作業をすれば、より040turboへの理解が深まると思います。

1.1.1　68030ソケットを介した信号

X68030はMPUの交換を想定した作りとはなっていないため、I/Oスロット等からはプロセッサを乗っ取ることはできません。また、I/OスロットはX68000の16ビット時代の

バスであるため、ここから信号をやりとりしていたのでは性能が上がりません。このため、040turboは、X68030のマザーボード上の68030（68EC030）を外し、そのソケットを介して必要な信号をやりとりします。このソケットとの接続は、図1.1(a)のように040turboから伸びている足を挿し込み、ドータボードの形で実装します。

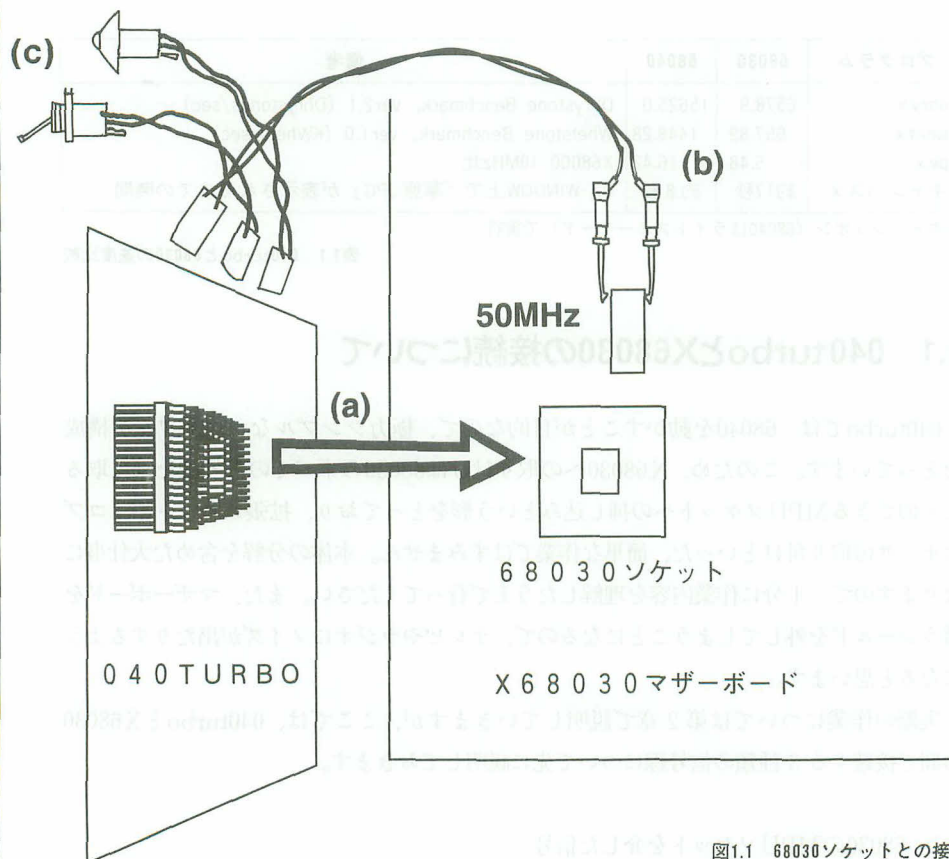



図1.1　68030ソケットとの接続

1.1.2　50MHzのクロック信号

68040は68030と異なり、2種類のクロックを必要としています。1つは、MPUの内部動作クロックとなるプロセッサクロック（PCLK）で、もう1つはメモリアクセスなどの外部バスとのタイミングの基準となるバスクロック（BCLK）です。プロセッサクロックはバスクロックの2倍で、信号の立ち上がりも揃っていなければなりません。

040turboは、68030の動作クロックである25MHzにあわせて25MHzのバスクロックで動作します。この場合、50MHzのプロセッサクロックが必要になりますが、もともとプロセッサクロックという信号が存在しない68030のソケットには、これは出力されてい

ません。このため、マザーボード上の50MHzのオシレータ出力から直接取り出す必要があります。実際にはオシレータの足がマザーボード表面から見えないため、オシレータ出力が接続されているIC（ここで、50MHzを分周して25MHzのクロックを作り出している）の足に図1.1(b)のようにICクリップ[*1]をかませて50MHzのクロックを直接取り出します。

なお、040turboの試作のとき、簡単な逓倍回路（信号を4分の1周期遅らせて、元の信号とEORすると、2倍の周期の波形になります。ただ、信号を正確に遅らせるのが、なかなか大変です）を作ってみましたが、動作が安定しませんでした。モトローラのMC88915という、クロックを2倍にする専用ICを使った倍クロック回路を試作したので、いずれはこの回路を搭載しようと考えています。

1.1.3　MPU切り替えスイッチと表示LEDの信号

040turboでの動作MPUの切り替えは、物理的なスイッチの切り替えで行います。また、動作MPUの表示はLEDに表示させるように信号を用意しています。040turboの基板自体は、X68030本体内部に隠れてしまうので、これらの信号は図1.1(c)のように外部に引き出します。

＊1
　マザーボードの50MHzオシレータ出力は、外部に取り出すことを想定した作りになっていないため、バッファ等は入っていません。また、マザーボードに改造を加えなくてもすむようにとICクリップという簡易な方法をとっているので、取り出された50MHzのクロック波形は相当に歪んだものとなっています。いちおう、040turboは動作しますが、ここがウィークポイントになっています。
　なお、クロックアップしたマシンでは安定動作をしない場合があります。そのときは配線を極力短くし、ICクリップを使わず、ICの足に直接半田付けしてください。

第2章
040turboの取り付け

X68030は外側カバーを外してもマザーボードはがっちりシールドに覆われています。コプロセッサに関してはシールドの部分だけ専用の穴が空けてあり、簡単に取り付けられるよう工夫がされていますが、040turboのように68030のMPUソケットに取り付けるドータボードについては、当然のことながら、考慮されていません。このため、040turboの取り付けは結構大変な作業になります。また、作業手順についての正式な手引きなども当然ありませんので、私の経験から、次のような手順を考えてみました。

- 前準備
- X68030のマザーボードが入っている右タワーの分解
  - (a)外側カバーの取り外し
  - (b)I/Oスロットの取り外し
  - (c)ビデオユニットの取り外し
  - (d)マザーボードの取り出し
  - (e)シールドの取り外し
- マザーボードの組み立て
  - (a)フレームの取り付け
  - (b)マザーボードの取り付け
  - (c)仮組み立てと動作確認
- ザーボードからの68030の取り外し
- 68030の040turboへの取り付け
- 50MHzクロック引き出し用ケーブルの取り付け
- 金属スペーサの取り付け
- 040turboの取り付け
- ケーブル接続
  - (a)50MHzクロック
  - (b)MPU切り替えスイッチ
  - (c)表示LED
- 040turboの動作テスト
  - (a)68030モード
  - (b)68040モード
  - (c)仮組み立てと動作確認

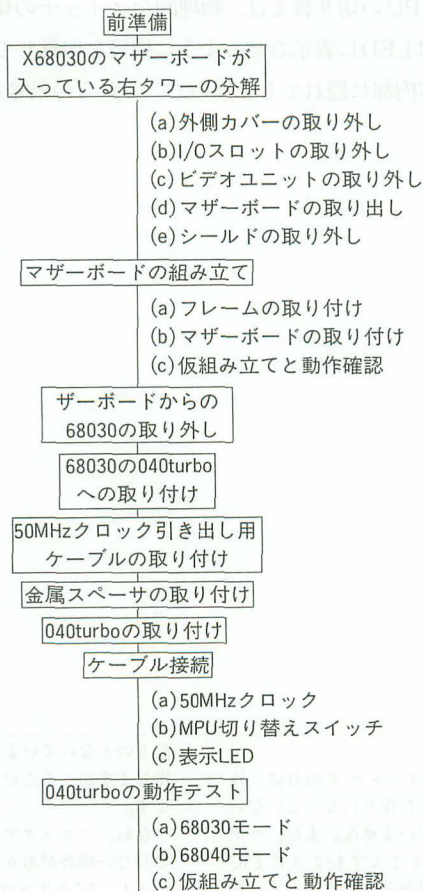



図2.1　040turbo取り付け作業の流れ

これらの作業を行った場合、メーカーによる保証はききません。また、もし改造作業によって、みなさんのX68030に障害を与えたとしても当方はいっさい責任をとれませんので、各自の責任で慎重に作業を行ってください。なお、いうまでもないことですが、以下の作業は電子回路を扱うので、静電気対策や作業環境などに十分注意して行ってください。

最初に断ったとおり、作業手順は私の取り付け経験によるものであり、100%成功の保証はありません。いきなり作業に取りかからず、ひととおり、この取扱説明書に目を通して、どんな作業があるのか、また、どんな点に注意すべきなのかを理解してから取りかかってください。すでにX68030を分解した経験のある人は、必ずしもこの手順どおりに進めなくてもかまいませんが、単に040turboをMPUソケットに挿すだけの作業ではありませんので、ご注意ください。

2.1 前準備

まず、040turboの添付品を確認してください。次のものが入っているはずです。

品名	個数
040turbo基板	1
50MHzクロック取り出し用ICクリップ付きケーブル	1
動作MPU切り替え用スイッチ付きケーブル	1
動作MPU表示用LED付きケーブル	1
基板固定用金属スペーサ	2
パッチプログラムのフロッピーディスク	1
取扱説明書	1

足りないようなら、連絡してください。

次に、取り付け作業のための工具を用意します。

X68030で使われているネジは基本的にプラスのドライバーで回せます。しかし、同じプラスのドライバーでも、先のとがったものから若干丸くなった太めのものまで数種類あります。X68030に使われているネジは太めのものです。ネジの十字形の溝とドライバーの先がピッタリあっていないと作業効率が悪いばかりでなく、ネジの溝をつぶしてしまうおそれがありますので、ネジの溝とあったものを用意してください。

また、柄の長いドライバーでは回しづらいところがありますので、全体の長さが10cmくらいの短いドライバーも用意しておくとよいでしょう。ラジオペンチは必須ではありませんが、ICの足の曲がりを整えるときに便利ですので用意しておくとよいでしょう。

さらに、マザーボードから68030を抜くとき、直径1mmくらいの精密ドライバーもしくはピンセットなどが必要になります。LSIを引き抜く専用の工具を用意できれば申し分

ありません。

最後に、Human68kのシステム一式の入ったブート用フロッピーディスクを用意しておいてください。これは、動作確認に使用します。また、メモリスイッチで起動ドライブを「STD」にしておいてください。マザーボードを外すときにメモリスイッチで設定した内容が消える可能性があるので、スイッチ内容等もついでに控えておくとよいでしょう。

2.2 X68030のマザーボードが入っている右タワーの分解

X68030は外側カバーがプラスチック製のため、VCCI対策としてマザーボードがシールドで覆われていますから、040turboを取り付ける際にはシールドを外してマザーボードのMPUソケットをむき出しにする必要があります。このとき、右タワー内をほぼ完全に分解することになりますので、次の手順に従って焦らずに慎重に作業を行ってください。当然ですが、コネクタや電源コードは必ず抜いてから作業に取りかかってください。

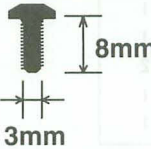
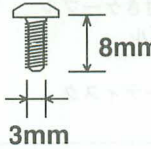
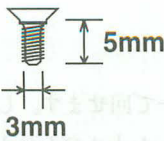
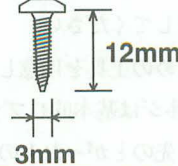
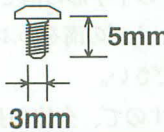
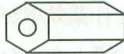
なお、これから図中で示すネジの番号とネジの外形のリストを図2.2に示しておきます。ネジの番号は、作業の順序にあわせてつけてありますが、必ずしも、このとおりにネジを外す必要はありません。

ネジの外形	ネジ番号	ネジの外形	ネジ番号
8mm 3mm	1～3	8mm 3mm	9～11 14～16 19～23
5mm 3mm	4	12mm 3mm	12 17、18
5mm 3mm	5～8 13 25～33		24

図2.2 X68030に使われているネジ

2.2.1　外側カバーの取り外し

まず、マザーボードが入っている右タワーの外側カバーを外します。図2.3に示した本体背面のネジ1～3を外して、カバーをずらします。カバーは前面パネルの接合部とツメで引っかかるようになっていますので、図のように、接合部を少し押しながらゆっくりずらしていきます。

なお、拡張メモリを増設してある場合はあらかじめ外しておいてください。

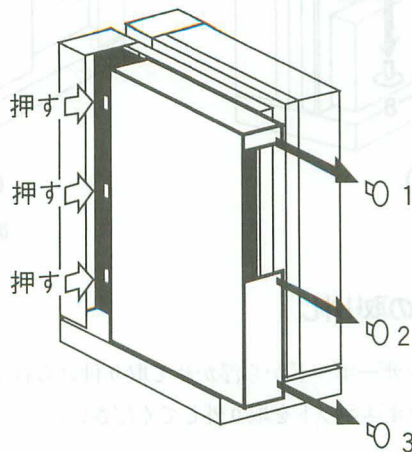



図2.3　外側カバーの取り外し

2.2.2　I/Oスロットの取り外し

外側カバーを外したら、次にI/Oスロットを外します。

I/Oスロットを包むシールドは必ずしも外す必要はありませんが、マザーボードのシールドと導通しやすいように金属の羽（というのかどうか正式な名称は知りませんが）が出ていますので、マザーボードのシールドを取った状態だと部品と接触しやすくなります。安全のため、I/Oスロットを外すだけでなく、それを包むシールドも外します。

図2.4(a)に示すネジ4～8を外すと、I/Oスロットのシールドが外れます。その後、図2.4(b)に示すネジ9～12を外してI/Oスロットを外してください。

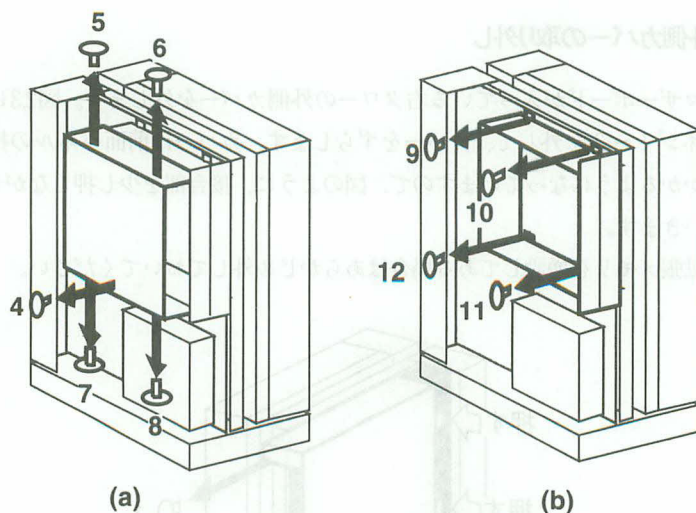



図2.4　I/Oスロットの取り外し

2.2.3　ビデオユニットの取り外し

ビデオユニットは、マザーボードから浮かせて取り付けられています。図2.5に示すネジ13*1〜16を外してビデオユニットを取り外してください。

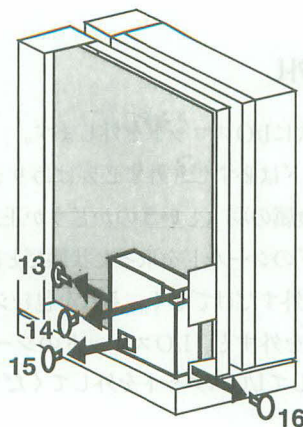



図2.5　ビデオユニットの取り外し

＊1
私のマシンでは13のネジでビデオユニットがシールドに留められていましたが、そうなっていないマシンもあるという報告をもらっています。13のネジを外さなくてもビデオユニットが外せるようなら、そのままでかまいません。

2.2.4　マザーボードの取り出し

マザーボードは、表側からは薄いシールドで覆われ、裏側からは補強の意味で少し厚めの金属フレームで保護された、サンドイッチ状態で本体に取り付けられています。したがって、マザーボードを本体に取り付けたままシールドだけを外すことはできません。いったん、本体からマザーボードごと取り出さなければなりません。図2.6に示すように、マザーボードの下につながっている2つのコネクタを外した後、ネジ17〜22を外します。

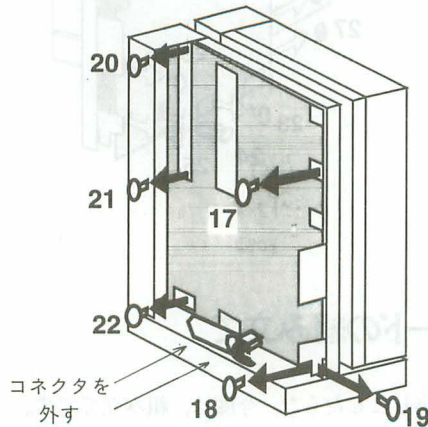



図2.6　マザーボードの取り出し

これでシールドごとマザーボードが取り出せるはずです。若干、引っかかるような手応えはありますが、特に力を入れなくてもよい*1はずなので、押しても引いても取れない場合は外し忘れて残っているネジがないかどうかを点検してください。マザーボードは本体とはつながっていないはずですが、私のX68030とネジ留めが変わっているかもしれません。

2.2.5　シールドの取り外し

マザーボードごとシールドを取り出したら、図2.7に示すネジ23〜33を外します。これで、マザーボードをサンドイッチ状にはさんでいたシールドと金属フレームが外れます。なるべく部品に触れないように注意しましょう。

＊1
さすがにX680x0シリーズも進歩しています。私のところには初代のX68000がありますが、最初にこのマザーボードを取り出すときは相当に無理な力を加えることになり、苦労しました。今は、前面パネルを外しているので取り出しは簡単です。ちなみに私は、前面パネルのネジを緩めるためにドライバーを1本、万力にはさんで90度曲げました。

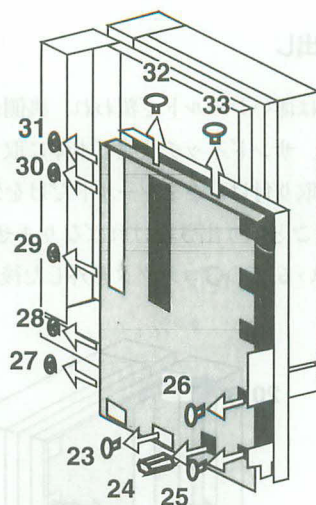



図2.7 シールドの取り外し

2.3 マザーボードの組み立て

マザーボードをむき出しにしたら*1、今度は、組み立てです。

基本的には分解の逆の順をたどるのですが、ネジ22、23は040turboを固定するために金属スペーサにかわりますのであわてて留めないように注意してください。

2.3.1 フレームの取り付け

まず、マザーボードを金属フレームにあわせ、シールドを外した状態で図2.7に示したネジ、24〜26を取り付けます。これらのネジは少し緩めにしておいてください。次にマザーボードを本体に取り付けますので、そのときに他のネジの具合を見ながら締めるようにしたほうが作業がしやすくなります。

ネジ23は040turboの取り付けで金属スペーサにかわりますので、ネジ留めをしないでください。ネジ23自体は後で使います。

表側のシールドを留めていたネジ27〜33は必要なくなりますが、紛失してしまうとシ

＊1
マザーボードとシールドの間は1cmくらいの隙間しかないので、040turboを搭載した上からシールドを取り付けることはできません。しかし、MPUソケットの真上にあたる部分のシールドに穴を空けて、その穴を通して040turboを装着することは可能かもしれません。ただ、高クロックのMPUがシールドの外に置かれることになるので、これでVCCI防止の効果があるかどうかは不明です。試してみた人の報告では多少の効果があるとのことですが、万全ではありません。

ールドを取り付けられなくなりますので、金属フレームのもとあった位置に仮留めしておくとよいでしょう。

2.3.2　マザーボードの取り付け

金属フレームに取り付けたマザーボードを本体にあわせ、319ページの図2.6で示したネジ17～21を取り付けます。

ネジ22は040turboの取り付けで金属スペーサにかわりますので、ネジ留めしないでください。

X68030は怪しげな互換機と違って、立て付けはしっかりしているので、ネジ穴があわないというようなことはありませんが、仮締めで全体を大まかに留めた後に増し締めするようにしてください。

マザーボードを取り付けたら、マザーボードの下につながる２つのコネクタも接続しなおします。

2.3.3　仮組み立てと動作確認

分解の手順とは逆に、ビデオユニット、I/Oスロット、拡張メモリの順に取り付けていきます。I/Oスロットを包むシールドはつけません。この時点では外側カバーおよびシールドがないだけで、X68030は正常に動作するはずです。

念のため、040turboを実装する前に、ここで動作確認をしてください。

電源、アナログRGBのコネクタ、キーボードという最低限の構成にし、前準備で用意しておいたブート用フロッピーディスクで正常に起動するかどうかを確認してください。

メモリスイッチの内容は初期化される可能性があるので、必ずしも分解前と同じとはかぎりません。慌てないようにしてください。

動作することが確認できたら、再度、メインスイッチを切り、コンセントを抜いて、次の作業にかかります。

注意

040turboを装着した後では、動作不良があったとしても、040turboの障害によるものなのか、分解・組み立て作業のミスによるものなのか、判別がつかなくなるので、面倒でも、この時点で必ず動作確認を行ってください。

もしも動作異常が発生したら、作業にミスがないかどうかよく確認してください。それでもダメならあきらめて修理に出したほうがよいでしょう。下手に作業を進めても傷口を広げるだけです。

2.4　マザーボードからの68030の取り外し

68030（68EC030）の取り外しは専用の工具を用いれば確実ですが、専用工具は結構な値段がします。X68030のソケットは、それほど堅くないので、先の小さなドライバー*1などで慎重にやれば取り外すことができます。一気に持ち上げるとMPUの足が曲がってしまいますので、いくつかの方向から少しずつ持ち上げるようにするのがコツです。

もし、ビデオユニットやI/Oスロットが邪魔になって無理な引き抜き方になりそうな場合は、面倒でも、これらを取り外してから作業を進めましょう。

2.5　68030の040turboへの取り付け

マザーボードから外した68030は、図2.8のように040turboの基板上のMPU１のソケットに取り付けます。取り付け方向を間違えないようにしてください。68030ならば１番ピンのある角に向かって金色の線が印刷されています。68EC030ならば１番ピンにあたる位置に凹みがあります。これをMPU１ソケットの１番ピンの位置にあわせて取り付けます。

図2.8に示した向きに040turboの基板を置いたとき、右上の角がMPU１ソケットの１番ピンの位置になります。040turboの基板上にはピンの行方向と列方向にアルファベットと番号が印刷されていますが、１番ピンはこのA行１列の位置に対応します。

ちなみに、68040は右下が１番ピン*2です。68040を別に用意した人、ファンの取り付けのために一度外した人は、挿入方向を間違えていないか、よく確認してください。

また、ソケットへの挿し込みはしっかり行ってください。挿し込みがゆるいと誤動作したり、最悪の場合、壊れ*3たりします。しっかり挿されば、横から見たソケットと、68030や68040の間の隙間は１mmくらいしか空かないはずです。

＊1
私は、先の曲がったピンセットを斜めから挿し込み、先の部分を支点にしてテコの要領で持ち上げています。

＊2
なぜ１番ピンの位置を統一しないんだ、という疑問もあるでしょう。実は、最初は68030も右下が１番ピンでしたが、X68030のマザーボードに取り付けるには68030の右上が１番でないといけないのです。これではX68030に取り付けできないわけで、このミスを急いで直すために68030の１番だけ位置を変えたのでした。

＊3
実際に68030が異常に発熱して壊れたという報告がありました。

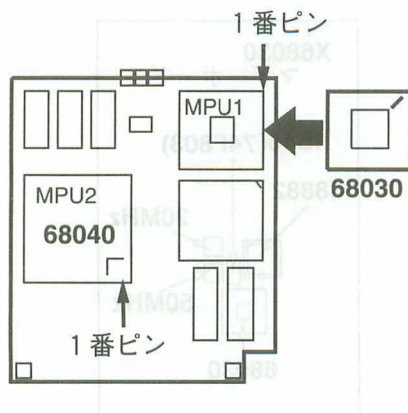



図2.8　68030の取り付け位置

なお、68EC030はピンが対称に植えられていて、間違った方向[*1]でも挿し込めてしまうので、十分注意してください。

また、68030の装着方向があっていても、ピンの数が多いので、ちょっとピンが曲がっただけで入りにくくなります。ピンが曲がっていたら、ラジオペンチなどでていねいに修正してからやりなおしてください。無理に押し込むと、ピンが完全に曲がってしまう恐れが十分あります。

2.6　50MHzクロック引き出し用ケーブルの取り付け

040turboの基板の実装位置が50MHzクロックを取り出すマザーボード上のICの上にきてしまいますので、後からではICクリップの取り付け作業が難しくなります。040turboの取り付けの前に、50MHzクロック引き出し用ケーブルを取り付けておきます。

取り付けるのは、部品番号IC46、部品名74F803という14ピンのICの７番ピン（GND）と８番ピン（50MHz）です。このICは、1993年８月現在出荷されているX68030のマザーボード上では、図2.9のように、MPUの上、コプロセッサ68882のソケットと50MHzのオシレータの間にあります。

＊1
間違った向きで電源を入れると、異常発熱し、ほぼ確実に壊れます。実際、これで68EC030を壊したという報告もありました。

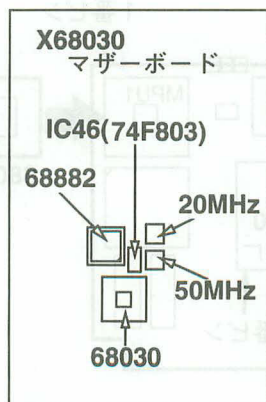



図2.9　50MHzクロックの取り出し位置

ただし、今後も74F803がこの位置にあるとはかぎりませんし、回路が変更される可能性もありますので、このICが見つからない場合は、部品名74F803をたよりに探してください。また、位置の変わったマザーボードがあった場合はその旨を報告してください。

取り付け位置を確認したら、付属の50MHzクロック引き出し用ケーブルについているICクリップを取り付けます。ICクリップは、図2.10(a)のように、先端の細長い部分が鉤の入った鞘になっており、鞘を引くと鉤が出てきます。この鉤をICのピンに引っかけ、鞘を離すと、バネの力で鞘が戻って鉤がピンに固定されるようになっています。

これで簡単に信号線を取り出すことができますが、信号の安定度からみると、よい方式ではありません。電子工作に自信がある人は、ICクリップを外して電線を直接ICの足に半田付けしてしまうほうがよいでしょう。

ICクリップの使い方がわかったら、マザーボード上のIC46に図2.10(b)のようにしてICクリップを取り付けてください。取り出す信号は50MHzクロックですが、信号を安定させるためにツイストペア線を使いますので、もう一方の線をGNDにつなぎます。ICクリップの赤（出荷ロットによっては白の場合もあります）が50MHz用、ICクリップの黒がGND用です。

ICクリップの取り付け位置にI/Oスロットがあるので、ちょっと作業がしづらいかもしれませんが、図2.10(c)のようにI/Oスロットの下から斜めに通して引っかけるようにすると、めったなことでは外れなくなるので確実です。ただし、むきになってICクリップをこじ入れると、マザーボードのパターンを傷めるので慎重に作業してください。

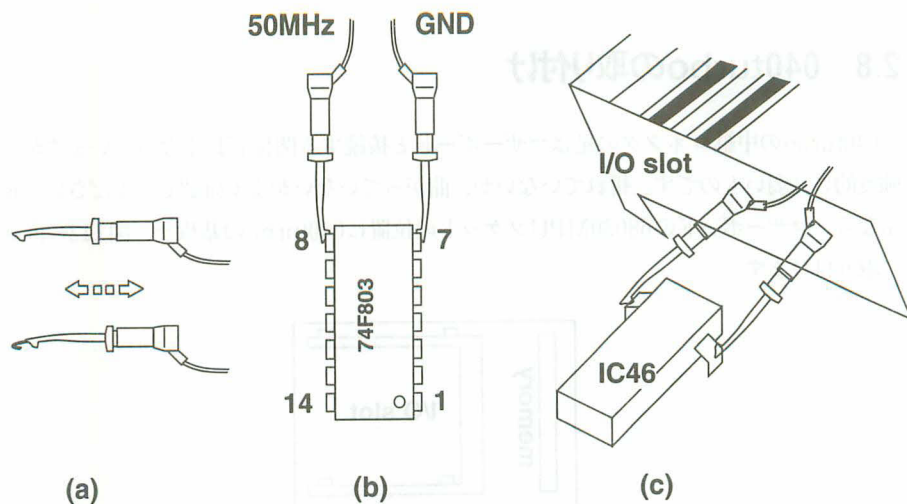



図2.10　ICクリップの取り付け

2.7　金属スペーサの取り付け

付属の金属スペーサを、図2.11のように、ネジ留めしないでおいたネジ22、23の位置に取り付けます。金属スペーサは、マザーボードに040turboを固定するとともに、GNDをつなぐ意味があります。

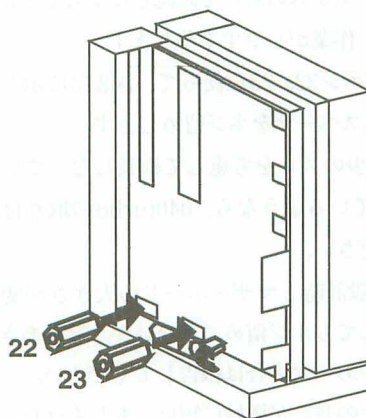



図2.11　金属スペーサの取り付け

2.8　040turboの取り付け

040turboの中継コネクタの足はマザーボードと接続する関係で長くなっていますが、強度的には弱いものです。折れていないか、曲がっていないかよく確認してください。正常なら、マザーボードの68030MPUソケットの位置に040turboの基板を、図2.12の向きに取り付けます。

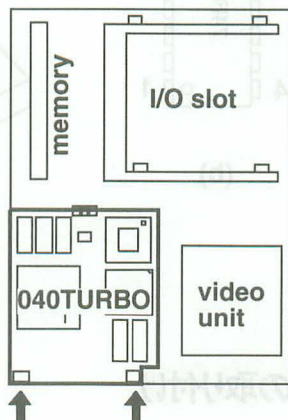



図2.12　040turboの取り付け

取り付けに際しては、マザーボードの68030MPUソケットの位置と040turboの中継コネクタの足の噛み具合を、横や上からのぞいて確認しながらしっかり挿し込んでください。ビデオユニットを取り外すと、作業がしやすくなります。

040turboを取り付けたら、ネジ22、23を使って、図2.12に示した矢印の2ヵ所で040turboと先ほど取り付けた金属スペーサをネジ留めします。

040turboの基板側の穴は多少のズレを考慮して横長になっていますが、金属スペーサのネジ穴とあまり大きくズレているようなら、040turboの取り付けがおかしい可能性があるので、見なおしてみてください。

なお現在では、040turboの設計時とマザーボードの大きさが変わっていることもありえますので、ズレていてどうしてもネジ留めできないようならあきらめてください。これも、そのようなマザーボードがあった場合は報告してください。

その他、付属の金属スペーサの長さが微妙に短い、もしくは長いという報告がありました。これは、ICソケットの長さがマシンによって若干異なることがあるからです。しっかり取り付けられない場合は、付属の金属スペーサの使用をあきらめ、適正な長さのものを使うようにしてください。

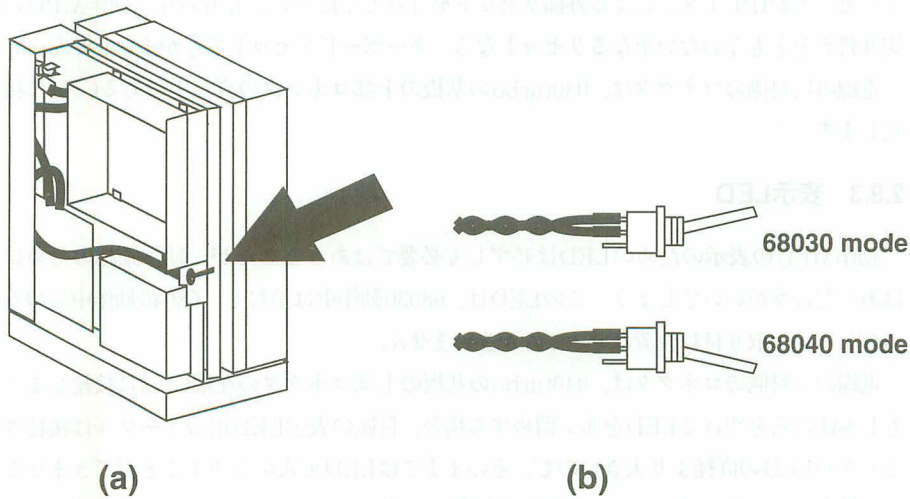



図2.13　MPU切り替えスイッチ

2.9　ケーブル接続

2.9.1　50MHzクロック

先に接続しておいた50MHzクロック引き出し用ケーブルのコネクタを040turbo基板の上部コネクタの右端、CN2に接続します。

2.9.2　MPU切り替えスイッチ

付属のMPU切り替えスイッチを適当な位置に取り付けます。たとえば、X68030の背面の図2.13(a)の矢印の位置にアース用の端子があるので、これを外した穴[*1]にスイッチを取り付けるのがお手軽でしょう。

スイッチは、図2.13(b)のように、接点が開いた状態が68030モード、接点が閉じた状態が68040モードです。

動作MPUの切り替えは、電源投入時のパワーオンリセット、および本体上部のIPLボタンによる外部リセット時のみ有効です。動作中にスイッチを切り替えても外部リセットがかかるまでは動作MPUが切り替わることはありません。

キーボードリセットなどのソフトウェアリセットでは、回路上、現在のMPUが動作したまま、もう一方のMPUも動作しようとしますのでハングアップします。必ずパワーオ

＊1
X68000のクロックアップ改造を生きがいとする知人の常套手段です。すでにクロックアップのための改造でこの穴を使ってしまっている人は別の手段を考えてください。

ンリセットかIPLボタンによる外部リセットを行ってください。もちろん、動作MPUの切り替えをともなわない単なるリセットなら、キーボードリセットでもかまいません。

電線の反対側のコネクタは、040turboの基板の上部コネクタの真ん中にあるCN1に接続します。

2.9.3　表示LED

動作MPUの表示のためのLEDは必ずしも必要ではありませんが、動作確認のためにはあったほうがいいでしょう。このLEDは、68030動作中は消灯し、68040動作中にのみ点灯します。取り付け位置はどこでもかまいません。

電線の反対側のコネクタは、040turboの基板の上部コネクタの左端CN3に接続します。もし本体に穴を空けてLEDをネジ留めする場合、付属の表示LED用のケーブルは接続コネクタがLEDの直径より大きいので、そのままではLEDを穴から通すことができません。この場合は、コネクタからコンタクト*1を抜いて通してください。

2.10　040turboの動作テスト

これでいちおう、040turboの取り付けは完了しました。動作テストを行いますが、もしテスターがあれば、電源を入れる前にVCCとGNDがショートしていないかどうか確認しておくとよいでしょう。040turbo基板上のIC1からIC3の20ピンのICは、10番ピンがGND、20番ピンがVCCです。この間の抵抗値が1Ω以下ならショートしている可能性があります。ショートしているようなら、040turboを外して、040turboだけで抵抗値を測ってみてください。もしここでショートしているようなら、040turboの基板自体の不良ですから連絡してください。

なお、68040は68030とは比べものにならないくらい熱を持つので、放熱対策が必須です。本格的な放熱対策は動作テストが完了してからでかまいませんが、68040が熱で壊れてしまうと元も子もないので、放熱対策をしていない場合、動作テストはできるだけ手短に行いましょう。ためしに動作中の68040に触ってみれば（火傷に注意！）、放熱対策の必要性を実感できると思います。なお、放熱対策についてはAPPENDIX Cを参照してください。

また，クロックアップされている方はウェイトを入れる必要がありますので、APPENDIX Eを参照してください。

＊1
コネクタ内部の、電線側の先の接点部分です。よく見ると抜けないように引っかかっている部分があるので、ここをシャープペンシルの先などで押すと電線といっしょにコンタクトを抜くことができます。コンタクトを挿し込むときはコネクタの位置を間違えないように注意してください。

2.10.1　68030モード

組み立て途中で動作テストしたのと同様に、電源、アナログRGBのコネクタ、キーボードという最小限の構成にし、前準備で用意しておいたブート用フロッピーディスクでテストします。

MPU切り替えスイッチを68030モードにして、電源を投入します。動作MPUの表示LEDは消えたままのはずです。通常どおりに起動し、各種のコマンドを実行して異常がないかどうかを確認してください。キー入力、FDアクセスがひととおりできればOKです。もし68030モードで起動できない場合は、040turboを外して、マザーボードに68030を直接取り付けて試してみてください。それでも起動できない場合*1は、どこか壊した可能性がありますので修理に出しましょう。68030をマザーボードに取り付けて起動できる場合は、もう一度040turboを取り付け直し*2、試してみてください。それでもダメ*3なら連絡してください。

2.10.2　68040モード

次に68040モードです。このモードでは、ROMDBがオンだと起動しませんので、もしオンに設定されている場合はオフ*4に戻しておいてください。

MPU切り替えスイッチを68040モードにしてIPLリセットするか、もしくは電源を入れなおします。キーボードリセットではハングアップしますので注意してください。今度は、動作MPUの表示LEDが点灯するはずです。点灯しない場合は、50MHzクロックの配線に問題がありますので見なおしてください。

68040モードでも、IPLの起動画面ではMPU68030と表示され、動作クロックを示す文字は変になりますが、これは動作異常ではありません。ブート用フロッピーで正常に起動するかどうか、各種のコマンドを実行して異常がないかどうかを確認してください。ただし、cache.xではキャッシュオンにできませんが、これは異常ではありません。逆に、キャッシュオンにできる場合は、68040モードになっていないということになります。キー入力、FDアクセスがひととおりできればOKです。

いちおうの動作確認がすんだら、外側カバーを取り付けて、組み立て完了です。

＊1
拡張I/Oスロットが外れていると、正常動作しないこともあるようです。

＊2
取り付けが甘いと動作しません。

＊3
出荷時にはすべてテストをしてから出荷するようにしていますが、輸送中に改造箇所等が断線することがあるかもしれません。

＊4
switch db＝off⏎でオフにできます。

第3章
040turbo用のプログラム

040turbo用のプログラムとして用意されているものは、X68030のプログラムを68040で動作させたときに問題になる部分に対処するパッチやユーティリティです。

68040は、デフォルトではキャッシュオフで動作します。また、68030のキャッシュ制御との互換性がないため、そのままではキャッシュオンになりません。このキャッシュオフ状態におけるHuman68kの動作は、ほぼ68030と同じと見てよく、実際、動作テストで見たようにパッチを施さなくてもなんら問題なく動作します。

しかし、68040はキャッシュオンを前提としたハードウエアのため、キャッシュオフでは本来のパフォーマンスを発揮できないだけでなく、040turboのハードウェアのオーバーヘッドも入るために68030より劣る性能しか出ません。キャッシュオンで動作するよう、パッチを当てるのは必須といえるでしょう。

ここでは、まずはじめに040turboを使用するうえでどのような不具合があるかを説明した後、対応プログラムについて説明します。

3.1 X68030の不具合

040turboをX68030上のHuman68kで使用するうえで現在、判明している不具合は、次のとおりです。

●MPU判定で68030と68040の判別がされない（IPL）

X68030のROMは、そもそも68040に対応していないので、68040かどうかを判別するためのコードがありません。現状では68030と判定されてしまいます。

[対処] 判定ルーチンを追加すべきですが、面倒なので結果を返す部分をパッチして決めうちで68040として返すようにしています。

●クロック測定が正常に実行されない（IPL）

X68030のIPL画面に表示されるクロック値は、インタバル割り込みを設定しておいて、キャッシュオンでループ命令を実行し、割り込みがかかるまでに何回ループを回ったかで計測しています。68030と68040ではキャッシュをオンにするコードが異なりますので、68040ではキャッシュオフのままで実行され、変な値になります。

[対処] キャッシュ制御コードを68040に対応するモードにパッチします。電源オン時や

IPLボタンによるリセットのときは対応できませんが、一度パッチした後のキーボードリセットでは正しく表示されるようになります。

●キャッシュオン時にI/Oアクセスが異常になる（IOCS）

キャッシュオン時はバースト転送が行われるため、I/Oに対するアクセスでよぶんなI/Oポートまでアクセスされてしまい、I/Oがおかしくなります。

[対処] MMUを使い、I/O領域をキャッシュ禁止でアクセスされるように設定しました。

●キャッシュ制御ができない（cache.x）

68040と68030は、キャッシュのオン・オフの方法が異なるため、68030のコードのままではキャッシュ制御ができません。

[対処] cache.xは新設されたIOCSコールであるIOCS-$AC［SYS_STAT］をコールしているだけなので、IOCS-$ACの処理を68040に対応するようにパッチをあてました。ほかのIOCS-$ACを使っているプログラムについてはキャッシュ制御できるようになります。

●キャッシュオン時の動作が異常（IPL、IOCS、Human68k、FSX、その他）

MPUがキャッシュオンで動作しているときにDMAやプログラムコードの書き換えなどをともなう処理をする場合は、キャッシュをクリアしなければなりません。キャッシュクリアの方法は68030と68040では異なっているので、68030のキャッシュクリアを行うコードのままでは68040のキャッシュはクリアされません。

[対処] 各プログラムの対応コードを68040に対応するようにパッチをあてます。キャッシュクリアはIOCS-$ACのファンクションとして用意されているので、通常のプログラムでは68030のキャッシュクリアのコードを使わず、じかにIOCS-$ACを呼び出すようにすべきでしょう。

●lzx化されたプログラムがエラーになる

これは、コード書き換えをしてもキュシュ上のコードは書き換えられずに残っているためと思われます。

[対処] patexec.sysがlzxの展開後にキャッシュクリアを行いますので、これを使用してください。それでも問題がある場合は、とりあえずキャッシュオフで使用してください。

3.2　68040対応プログラム

現在、040turbo用に以下のプログラムを用意しています。ここでは、使用方法を中心に説明します。具体的な内容については、第5章を参照してください。

1. 040SYSpatch.x

ROM内のIPL、IOCS、割り込み処理と、Human68k、SCHDISK（format.xが書き込むSCSIデバイスドライバ）のキャッシュ制御コードに対してパッチを当てます。

xの拡張子を持っていますが、デバイスドライバの形態で組み込み、起動時にメモリ上のコードに対して書き換えを行います。

本プログラムは、私のプログラムに、じゃぎゅあ氏ならびにPUNA氏によって修正が施されたものです。

2. patexec.sys

中村ちゃぷに氏作のダイナミックパッチャの68040暫定対応版です。

プログラム名とパッチデータを記述したファイルをもとに指定プログラムがメモリ上にローディングされると、メモリ上のコードに対してパッチを当ててくれます。次のプログラム用のパッチが用意されています。

fastio.x、fsx.x、cddev.sys

3. 040MPU.x

動作MPUを示すユーティリティ。単なる動作MPUの確認のほか、バッチファイル中で動作MPUによって処理内容を変える場合などに活用できます。

4. 040cache.x

68040のキャッシュモードの表示と設定を行うためのユーティリティです。68040では、68030と同様のライトスルーモードのほかに、コピーバックモードというキャッシュの動作モードを持っています。コピーバックモードでは、データ書き込みはキャッシュ上でのみ行われ、必要になるまで実際のメモリへの書き戻しが行われないので、より高い性能を期待できますが、互換性に関しては問題が生じる可能性があります。

なお、040cache.xはフリーエリアのキャッシュモードを切り替えるだけです。キュッシュ自体のオン、オフは、Human68k付属のcache.xを使用してください。

5. allcache.x

じゃぎゅあ氏作のキャッシュモード設定ユーティリティです。040cache.xがフリーエ

リアのみを対象としているのに対し、全メモリを指定のキャッシュモードに設定します。

6. setcache.x

Y.さん作のキャッシュモード設定ユーティリティです。アドレス範囲を指定してキャッシュモードを設定することができます。

7. float040.x

鈴木国文氏作のfloat演算パッケージです。68040の浮動小数点演算命令は、68882のサブセットになっていますので、float4.xを使用することはできません。float2.xでは性能が出ませんので、68040の浮動小数点演算命令を使用してコーディングされたものが、このflaot040.xです。float2.xやfloat4.xのかわりに使用することで、float4.xよりも高速に動作します。

8. pfloat.x

中村ちゃぷに氏作の疑似コプロエミュレータです。68040の浮動小数点命令は、68882のサブセットになっており、68030と68882の組み合わせを想定して作られたプログラムでは、不都合が生じます。このプログラムを登録しておくと、68040でサポートしていない浮動小数点演算命令については、マザーボード側の68882を使用してエミュレートしてくれます。

9. GCC

まりこさん作のGCCです。68040に対応したコードを出すオプション（-m 68040）が追加されています。

040turboの配布によって、多くの人の手によってプログラムの対応作業が行われていますが、まだ問題が発生するプログラムもあります。また、プログラムがバージョンアップすれば、パッチもバージョンアップする必要があるでしょう。68040対応プログラムの最新情報については、NIFTY-ServeのFSHARP 1（SHARPユーザーズフォーラム）の「ハードウェアの部屋」をアクセスしてください。

次に、主なプログラムの使用方法について説明します。

3.2.1　システムへのパッチ040SYSpatch.x

このプログラムは起動時にMPUが68040かどうかを判定し、68040である場合にはROM内のIPL、IOCS、割り込み処理とHuman68k、SCHDISKのキャッシュ制御コードに対しパッチを当てます。68030の場合はパッチを当てませんので、つねに組み込んだ

状態にしておいて支障ありません。

ROM内のIPL、IOCS、割り込み処理に対しては、$FF0000～$FFFFFFの内容をRAMにコピーしてから必要なパッチを当てます。対応するROMは、IOCS-$8F［ROMVER］で$13921127(16進)が返る'92年11月27日付けのものだけです。'94年4月現在、他のバージョンの存在は報告されていません。ROMのバージョンが違うマシンがあったら報告してください。

Human68kおよびSCHDISKに対しては、起動しメモリ上に展開されたコードにパッチを当てます。対応するHuman68kは、ver3.01、ver3.02＊1です。SCHDISKは、SCSIDISK DRIVER for X68000 version 1.04です。

プログラム中でもバージョンチェックやパッチコードのチェックを行っています。異なるバージョンの場合はエラーメッセージを表示して、パッチをスキップします。SCSIのデバイスドライバについてはパッチに失敗しても実害はないようですが、念のため、フォーマットしなおして対応バージョンのSCSIデバイスドライバに差し替えることをお勧めします。Human68kのバージョンの違いによってパッチが失敗した場合は危険です。たとえ起動して正常のように見えても、そのまま使用するのは避け、68030モードに戻して対応するバージョンに差し替えるようにしてください。

使用方法

このプログラムは、デバイスドライバの形で登録します。システム自体へのパッチという性格上、最初に登録されるように必ずconfig.sysのdevice文の先頭に記述してください。コマンドラインから実行した場合はバージョン番号を表示するだけです。パッチに必要なROMのコピー領域やワークエリアは、Human68kのメモリ管理情報（1C00番地）を直接操作して、メモリの最高位番地から確保しています。

例）040SYSpatch.xの登録

config.sysに以下の1行を書き加えます（040SYSpatch.xがディレクトリ¥sys¥にある場合）。

```
device = ¥sys¥040SYSpatch.x
```

040SYSpatch.xにはいくつかのオプションがありますが、とりあえず何もつけなくてかまいません。オプションの詳細については添付ディスクのドキュメントを参照してください。

＊1　DOS-FF30［vernum］で$36380301（16進）か$36380302（16進）が返るもの。

config.sysへの040SYSpatch.xの記述を完了したら、リセットしてください。

Human68kが起動し、ROMの内容をRAMにコピーしたうえでパッチします。Human68k、SCHDISKにパッチが当てられます。

なお、このパッチは最低限のことしか行っていません。実際、いくつか問題も見つかっています。また、ROMやHuman68kのバージョンが上がっていけばパッチが対応しきれなくなるでしょう。

ROMやHuman68kのバージョンアップがあった場合は可能なかぎり対応に努めますが、完全とはいえません。ある程度のリスクを覚悟のうえで使用してください。また、パッチがどんなことをやっているのかについては第5章で説明していますので、ぜひ、一度目を通して内容を理解してください。

3.2.2　ダイナミックパッチャpatexec.sys

このプログラムも、デバイスドライバの形で登録します。あらかじめパッチ用の定義ファイルを用意しておく必要があります。

例）添付のパッチ定義ファイル040.patを使用する場合

1. パッチファイルの記述

最初にパッチ対象のファイル名を指定します。68030、68040で別のパッチを施すことも可能です。68040に対するパッチの場合は、次のように対象ファイル名の前に"4"を書きます。

```
-4-fastio.x
```

後は、次のようなパッチデータが続きます。

```
00001234:  56 78
00009ABC:  DE F0
```

パッチアドレス、旧データ、新データの順です。この書式は、fc.xのバイナリデータ比較時に表示されるものと同じです。添付の040turbo.patを参考にして、パッチデータを追加していくようにすればよいでしょう。

2. config.sysの記述

config.sysに以下の1行を書き加えます（patexec.sysおよび040turbo.patがディレクトリ¥sys¥にある場合）。

```
device = ¥sys¥patexec.sys ¥sys¥040.pat
                          ↑
                          └─ パッチファイル名
```

パッチファイル名を省略した場合は、デフォルトとして"¥etc¥patfile"が使用されます。

添付の040.patでは、FSX.X（SX-WINDOWシステム）およびfastio.x、cddev.sysに対するパッチを行っています。

3.2.3　動作MPUの表示040MPU.x

このプログラムは、動作MPUを標準出力に出力するとともに終了コードを返します。MPUによって処理を変えるときに使えるでしょう。

使用方法

040MPU.xを実行すると、次のようにMPU名を表示します。

```
MPU-68040
```

動作MPUと表示および終了コードとの関係は、次のとおりです。

動作MPU	標準出力	終了コード
68000	MPU-68000	0
68010	MPU-68010	1
68020	MPU-68020	2
68030	MPU-68030	3
68040	MPU-68040	4

3.2.4　動作キャッシュモードの表示と設定040cache.x、allcache.x、setcache.x

これらのプログラムは、68040のキャッシュモードの表示および設定を行うプログラムです。Human68k付属のcache.xによりキャッシュオンにしても、デフォルトではライトスルーモードのキャッシュしか有効になりません。040cache.xは、プログラム実行時点で使用可能な全フリーエリアを指定されたキャッシュモードに設定します。引数なしで実行した場合は現在のキャッシュモードを表示し、引数でキャッシュモードを指定した場合は、プログラム実行時点で使用可能な全フリーエリアを指定されたキャッシュモードに設定します。

allcache.xはメインRAMのエリアすべてのキャッシュモードを指定できます。高速動作が期待できますが、トラブルも起きやすいようです。setcache.xは040cache.xの機能に加え、任意のアドレスを指定することもできるように拡張されたものです。

指定できるキャッシュモードは次のとおりです。

引数	キャッシュモード
0またはw	ライトスルーキャッシュ
1またはc	コピーバックキャッシュ
2またはs	キャッシュオフ(シリアライズドアクセス)
3またはn	キャッシュオフ(ノン・シリアライズドアクセス)

シリアライズドアクセスは、命令の並び順にアクセスが実行されるモードです。040SYSpatch.xでは、I/O領域にこのモードを設定しています。通常は使用する必要はないでしょう。

なお、内部的に68040のMMUの機能を用いてキャッシュモードを設定しているので、設定の境界は8Kバイト単位になっています。

例) 現在のキャッシュモードを表示する

引数を指定せずに実行すると、次のように現在のメモリマップに対するキャッシュモードを表示します。

```
address    cache-mode
00000000 …0 (write through)
00200000 …1 (copy back)
00BF0000 …0 (write through)
00C00000 …3 (non cache)
00E80000 …2 (serialized non cache)
00EB0000 …3 (non cache)
00EC0000 …2 (serialized non cache)
00ED0000 …3 (non cache)
00EE0000 …2 (serialized non cache)
00F00000 …0 (write through)
```

なお、このプログラムは040SYSpatch.sys ver2.1c以降のバージョンでのみ有効です。

3.2.5 浮動小数点演算パッケージ float040.x、pfloat.x

float040.xは、動作MPUが68040かどうかをチェックして、68040でない場合は登録されないようになっているので、float2.xやfloat4.xの前に書いておけば68030モードとconfig.sysを共用することができます。

例) 68040ではfloat040.x、68030ではfloat4.xを使用する

config.sysを以下のようにします(各floatパッケージがディレクトリ¥sys¥にある場合)。

```
device = ¥sys¥float040.x
device = ¥sys¥float4.x
```

68040の場合はfloat040.xが登録され、float4.xはすでにfloatパッケージが登録されているので登録がスキップされます。68030の場合はfloat040.xがスキップされ、float4.xが登録されます。

pfloat.xは常駐プログラムですので、実行するだけでOKです。以後、68030+68882をターゲットにしたプログラムもエラーにならずに実行できるようになります。

3.3　注意事項

最後に、040turboをX68030で使用するための手順をまとめておきます。

1. ROMDBはオフにしておく
2. 040SYSpatch.xを登録する
3. patexec.sysを登録する
4. float040.x、pfloat.xを必要に応じて登録する
5. autoexec.batなどのバッチのなかで68030と68040の処理を分けたいときには040MPU.xの終了コードで振り分ける
6. cache.xでキャッシュをオン・オフする
7. 040cache.x、allcache.x、setcache.xなどを使ってシステムにあわせ、ライトスルー、コピーバックのキャッシュモードを指定する。

なお、これらのプログラムはフリーソフトウェアです。不具合や事故があっても、プログラム作者は責任を取らないことを、あらかじめご了承ください。

実際、まだ完全にテストされていない状態ですので、なんらかの不具合は出るでしょう。68030モードでファイルのバックアップを取ってから使用するようにしてください。

第４章
040turboハードウェア説明

68040のバスは68030のバスと同じ32ビットですが、その制御方法はかなり変更されています。68030では68000の時代から受け継がれてきた非同期アクセスと呼ばれるアクセス方式*[1]をとっているのに対し、68040では同期アクセス方式となっています。

また、ほかにも多くの違いがありますが、これらを040turboのハードウェアが吸収するようになっています。ただし、68030と68040の完全互換性が実現されているわけではなく、X68030で必要となる最低限の部分のサポートしか行っていません。まだ見つけられないままの問題が残っているかもしれません。

この章では、68030と68040の相違、および、040turboのハードウェアについて説明します。

APPENDIX Dに040turboのハードウェア関連の図面やGAL*[2]のソースを添付しました。まだ情報としては不足していますが、可能なかぎり質問等に対応しますので、040turboのハードウェアの問題点の洗い出しや改善のためにご協力をお願いします。

4.1　68040と68030の相違

68040は、68030とハードウェアレベルで、次のような相違点があります。

1. クロック
2. バスアクセス
3. ダイナミックバスサイジング
4. バスアービタの動作
5. 転送属性信号
6. その他の信号線の動作

以下、細かく見ていきましょう。

＊１
68030でも同期アクセスの機能を持っていますが、X68030のメモリコントローラは同期アクセスをサポートしていないようです。なお、同期・非同期とは、クロックに対してアクセスが同期しているかいないかということです。

＊２
General Array Logicの略称。動作内容をプログラムすることが可能なICの１つです。専用のGALライタを使って書き込みます。

4.1.1　クロック

前に説明[*1]したように、68040では68030と同様のバスアクセスにするため、基準クロックをバスクロックとし、新たにその2倍の周波数のプロセッサクロック[*2]で内部を動作させるようになりました。

しかし、68030のソケットにはバスクロック相当のクロックしか出力されていないため、040turboではマザーボードの25MHzクロックの元となっている50MHzのオシレータから直接プロセッサクロックを取り出すようにしています。

実際には、68040の信号がバスクロックの立ち上がりエッジで有効なのに対し、68030の信号が主にクロックの立ち下がりで変化することから、IC6によって68030のクロックを反転した信号を作り、68040のバスクロックとして与えています。

各クロックの関係をまとめると、図4.1のようになります。

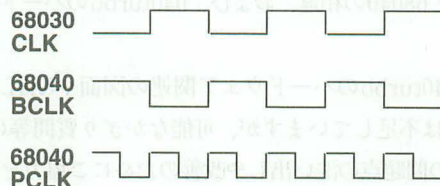



図4.1　68030と68040のクロックの関係

4.1.2　バスアクセス

68030では、図4.2のように、$\overline{\mathrm{AS}}$、$\overline{\mathrm{DS}}$信号のアサート[*3]によりバスサイクルが開始[*4]され、周辺回路はリード時ならデータバス上にデータを出力し、ライト時ならデータバス上のデータを取り込んだ後、$\overline{\mathrm{DSACK0}}$-$\overline{\mathrm{DSACK1}}$信号より応答するという方式をとっています。

バスエラー発生時は、$\overline{\mathrm{DSACK0}}$-$\overline{\mathrm{DSACK1}}$信号のかわりに$\overline{\mathrm{BERR}}$信号をアサートするようにします。アクセス中は$\overline{\mathrm{AS}}$、$\overline{\mathrm{DS}}$信号はアサートされ続け、応答を受け取ると、バ

＊1
「1.1.2　50MHzのクロック信号」

＊2
インテルの80486DX2やオーバードライブプロセッサと同様の発想でしょう。ただし、インテルのチップがクロックダブラという回路を搭載して外部のバスクロックを内部で2倍にしてくれるのに対し、68040は外から2倍のクロックを与えなければなりません。

＊3
信号が有効になることを「アサート」と呼びます。逆に、信号が無効になることを「ネゲート」と呼びます。$\overline{\mathrm{AS}}$のように、信号名の上にオーバーラインがついたものは、電圧のLowレベルが有効の意味になる負論理信号です。

＊4
実際にはAS信号の前に、アドレス信号とともにECS、OCS信号が先行して出力されますが、X68030はECS、OCS信号を使っていないので、040turboではサポートしていません。

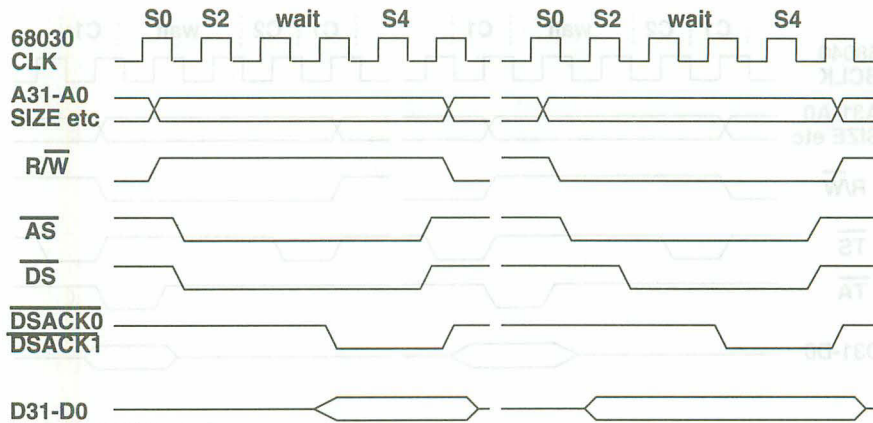



図4.2 68030のバスサイクル

スサイクルが終了されるときにネゲートされます。周辺回路は、AS、DS信号がネゲートされるまで、DSACK0-DSACK1信号をアサートし続け、AS、DS信号がネゲートを確認してDSACK0-DSACK1信号をネゲートします。

68030のバスアクセスは、これらの信号線でのみバスアクセスが決定され、クロックとはまったく関係しないことから、「非同期バス」と呼ばれています。とはいっても、実際には、68030がDSACK0-DSACK1信号を受け付けて1クロック後にバス上のデータを取り込むなど、細かなタイミングについてはクロックとの関係で決まっています。高速に動作させるためには、データが揃う前にDSACK0-DSACK1信号を先行して返しておけばよいわけです。68030のユーザーズマニュアルでは、これを「DSACKxとの同期動作」と呼んでおり、X68030でも先行してDSACK0-DSACK1信号が返っていますから、完全な非同期バスになっていません。

68040の同期アクセスは信号線自体が68030とはまったく異なっており、各信号は完全にバスクロックに同期して動くようになっています。具体的には、図4.3のように、TS信号のアサートによりアクセスが開始され、周辺回路からの応答はTA信号により行われます。TS信号自体は、サイクルの最初のクロックの立ち上がり部分でのみ有効であり、次のクロックではネゲートされます。また、応答する周辺回路側もTA信号もクロックの立ち上がりで有効になるように返さなければなりません。

エラー発生時は、TA信号のかわりにTEA信号をアサートするようにします。

このように、68030は周辺デバイスのためにバスの面倒をよく見てくれますが、68040

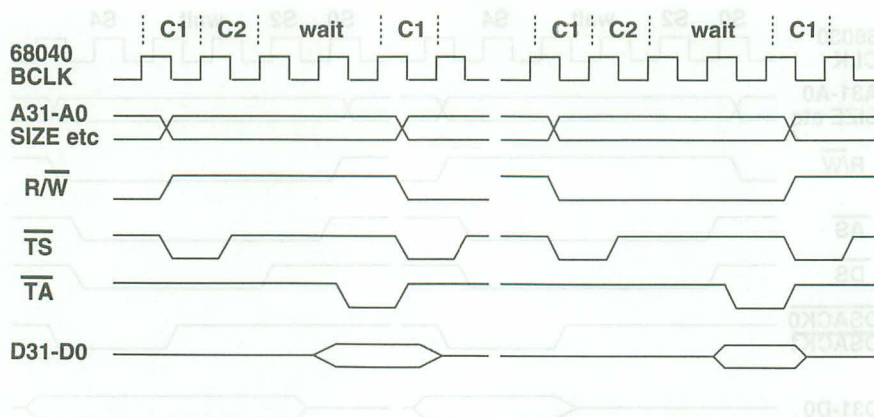



図4.3　68040のバスサイクル

では突き放し*[1]たような制御になっています。

040turboでは、68040と68030とのバスアクセスの違いを吸収しています。

68040の信号の変換部

この部分の働きは、68040の$\overline{TS}$信号から68030の$\overline{AS}$、$\overline{DS}$信号を作り出すことです。

```
tsn         = TS # next_bus & !xta;
tsn_wait.d = tsn;
new_tsn     = tsn_wait  & wait_sw
            #tsn       & wait_sw;
AS.d        = new_tsn   & !xta
            #AS        & !xta ;
DS.d        = new_tsn   & !xta & read
            #AS        & !xta ;
```

これはIC1のなかで行われています。IC1はGALというプログラム可能なICで、CUPLというGALコンパイラプログラムの次のソースで動作ロジックが記述されています。

ここで、簡単にCUPLのソースの読み方を説明しておきましょう。

各信号は、入出力ピンの定義で正論理・負論理を指定するので、論理式上はすべて正論理として扱うことができます。

"#"はOR、"&"はAND、"$"はXOR、"!"はNOTを表しています。".d"がついている信号はクロック信号の立ち上がりで取り込まれるフリップフロップ出力です。また、ここにはありませんが、.dがつかない信号は論理式の結果がそのまま出力されます。

＊1
実際、この形容は当たっています。68030ではバスの面倒見がいいかわりに、バスアクセスの間、他の仕事をあまりしません。これに対し、68040では突き放した後、内部でせっせとほかの仕事をしています。このため、バスを観察していると、68030ではアクセスとアクセスの間に隙があるのに対し、68040では次から次へとバスアクセスをしていきます。

IC1では、X68030のクロックを反転したバスクロックをつないでいますので、040turboによるAS、DS信号は、new_tsnが有効になったとき、X68030のバス上にはクロック信号の立ち下がりにあわせてアサートされ、以後はxtaが有効になるまでの間アサートされ続けます。

さて、このnew_tsnは最初の設計時にはなかったのですが、実回路で試した結果、ウェイトを入れられるようにしたものです。tsn_wait.dはフリップフロップ出力なので、tsnより１クロック遅れます。このtsnとtsn_waitのどちらを使うかをwait_swで選択できるようになっています。

next_busは、ダイナミックバスサイジングで使用されるサイクルのためのものです。これについては、4.1.3で説明します。

xtaは、IC2から返されるバスサイクルの終了信号です。ここでは、TAもしくはTEA相当と思ってよいでしょう。

68030の信号の変換部

この部分の働きは、68030のDSACK0-DSACK1信号やBERR信号から、68040のTA、TEA信号を作り出すことです。IC1およびIC2のなかで、次のようなロジックで行われています。

まず、IC1ですが、ここではDSACK0-DSACK1の受け付けをしています。

IC1のなかのロジック

```
dsa_wait.d = ( DSACK0 # DSACK1 ) & AS ;

d_long     = ( DSACK0 & DSACK1 ) & AS & dsa_wait;

d_word     = ( DSACK0 $ DSACK1 ) & AS & dsa_wait;
```

dsa_waitは68030がDSACK0-DSACK1の認識をクロックの立ち下がりで行うことに対応させたもので、IC1のクロックは68030のクロックを反転させたものですから、dsa_waitがアサートされるのはちょうど68030が立ち下がったときとなります。

d_long、d_wordは、DSACKxの応答とアクセスした周辺回路のバスサイズをデコードしたものです。これを受けて68040への応答を作り出すのがIC2です。

IC2のなかのロジック

```
s_word = !s_long;
as_mask.d = AS;
dsa_wait.d = ( d_long #d_word );
wait_mask = !wait_sw #dsa_wait;
```

```
xta.d      = AS & ( ( d_long # d_word ) & wait_mask
             # (BERR # AVEC) & as_mask );
ta_sub.d = AS ( s_word & d_word # s_long & d_word & next bus ) & wait_mask;
TA.d       = ( ( d_long & wait_mask ) # ta_sub ) &! AVEC # as_mask & AVEC;
TEA.d      = AS  & BERR & as_mask;
DLE        = AS;
```

IC2のクロックは、68030と同じ位相のクロックです。

s_longはSIZ0-SIZ1信号をIC3でデコードしたもので、現在のサイクルがロングワードか否かを示しています。

as_maskは、主にバスエラーの検出を1クロックの間、マスクするためのものです。040turboでグラフィックVRAMをアクセスすると、次のサイクルへと遷移するときのアドレス線の信号変化でX68030がバスエラーを出してくるという障害がありました。これに対処するために、サイクルの最初ではバスエラーを検出しないようにマスクしているのです。なお、68030でバスエラーが出ないのはアドレス変化のタイミングが異なることが根本原因のようですが、ほかにもVRAMにゴミが出るなどの問題が起こったため、今はVRAMなど16ビット系の周辺回路をアクセスする場合、68040のDLEモードという特殊なモードを使ってアドレス変化のタイミングを遅らせています。このためにta_subが使われています。

xtaは、DSACK0-DSACK1信号もしくはBERR、AVEC信号など、バスサイクルが終了することを示す信号です。IC1は、これを見てバスサイクルを終結させます。

AVEC信号は、68030のオートベクタ割り込みのアクノリッジ信号です。68040でも同名の信号がありますが、68040ではこの信号だけでなく、TA信号も返さないとオートベクタ割り込みのアクノリッジになりません。

TA信号は、68040に対する応答信号です。d_longがアサートされている場合は32ビット系なので、wait_maskを待ってTAがアサートされます。wait_maskはwait_swが無効の場合はつねに1なので、TAはウェイトが入らずにアサートされます。wait_swが有効な場合は、dsa_waitがアサートされるまで、すなわちd_longが1クロック遅れてアサートされるまで、TAが待たされるのです。DSACK0-DSACK1信号の応答がロングワード（d_long）でない場合はta_subで決まります。基本的には、DSACK0-DSACK1信号もしくはAVEC信号が返ってきたらTA信号をアサートすると思ってよいでしょう。

TEA信号は、BERRが返ってきたら、TEA信号をアサートします。

040turboの信号

68040の信号変換ロジックと68030の信号変換ロジックにより、040turboのバスアクセスは図4.4のように行われます（周辺回路が32ビットサイズの場合）。

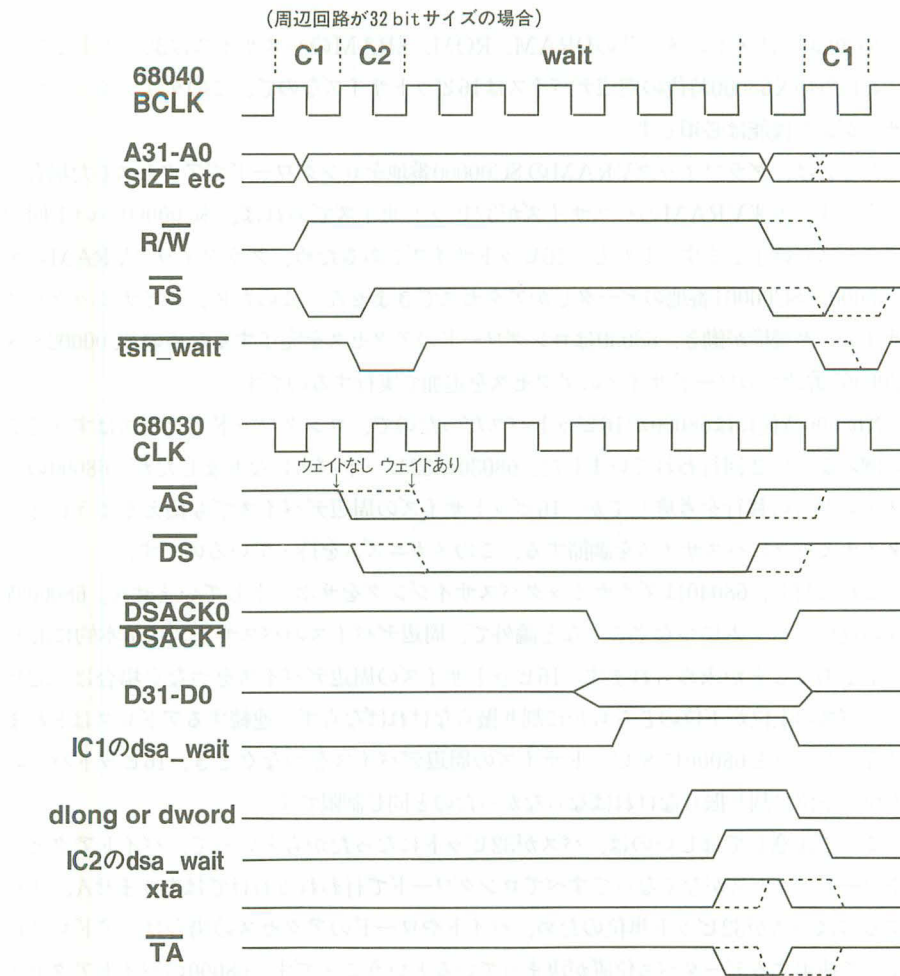



図4.4　040turboのバスサイクル
点線はウェイト時の動作。

4.1.3 ダイナミックバスサイジング

ダイナミックバスサイジングとは、周辺デバイスから応答といっしょにバスサイズを返すようにし、プロセッサ側でバスサイズにあわせてアクセスのしかたをダイナミックに変更する機能です。

68030はデータバスが32ビットですが、周辺デバイスは8ビット、16ビット、32ビットのどれでもかまいません。周辺デバイスは、自分のバスサイズをDSACK0-DSACK1信号で示し、68030はこのDSACK0-DSACK1信号の組み合わせを見て、バスサイズがアクセスすべきデータのサイズよりも小さいとき、残りのデータをアクセスするための追加のバスサイクルを再度実行します。

X68030ではメインメモリのDRAM、ROM、SRAMのバスサイズは32ビットですが、これ以外のX68000時代の周辺デバイスは16ビットサイズなので、このダイナミックバスサイジング機能は必須です。

たとえば、グラフィックVRAMの$C00000番地をロングワードでアクセスした場合を考えます。本来VRAMのバスサイズが32ビットサイズであれば、$C00000への１回のアクセスで終了します。しかし、16ビットサイズであるため、グラフィックVRAMの$C00000～$C00001番地のデータしかアクセスできません。このため、ダイナミックバスサイジング機構が働き、68030はロングワードのアクセスを完了するための$C00002～$C00003番地へのワードサイズのアクセスを追加で実行するのです。

X68000時代には68000が16ビットバスだったので、ロングワードアクセスはすべてこの例のように２回行われていました。68030は32ビットバスになりましたが、68000のシステムからの移行を考慮してか、16ビットサイズの周辺デバイスでも使えるようにと、ダイナミックにバスサイズを調節する、このメカニズムを持っているのです。

これに対し、68040はダイナミックバスサイジングをサポートしていません。68000時代の遅いシステムにつなぐことなど論外で、周辺デバイスのバスサイズは基本的に32ビットであることが求められます。16ビットサイズの周辺デバイスをつなぐ場合は、32ビットバスの上位か下位のどちらかに割り振らなければならず、連続するアドレスはとれません。ちょうど68000に８ビットサイズの周辺デバイスをつなぐとき、16ビットバスの上位か下位に割り振らなければならなかったのと同じ制限です。

ここで注意してほしいのは、バスが32ビットになったからといって、バイトアクセスやワードアクセスがなくなってすべてロングワードで行われるわけではありません。注意するのはバスが32ビット単位のため、バイトやワードのアクセスの場合は、アドレスによって使用するデータバス位置が決まっているということです。68000のバイトアクセスで、偶数番地と奇数番地がそれぞれデータバスの上位と下位で行われていたのと同じと考えればわかりやすいでしょう。

68040では、アクセスするサイズとアドレスによって、使用されるデータバスが次のように決まっています。

SIZE	A1	A0	D31〜D24	D23〜D16	D15〜D8	D7〜D0
long	0	0	使用	使用	使用	使用
word	0	0	使用	使用	——	——
word	1	0	——	——	使用	使用
byte	0	0	使用	——	——	——
byte	0	1	——	使用	——	——
byte	1	0	——	——	使用	——
byte	1	1	——	——	——	使用

これに対し、ダイナミックバスサイジング機構を持つ68030では、ワードサイズの周辺デバイスはつねにデータバスの上位D31~D16を使い、バイトサイズの周辺デバイスはデータバスの最上位D31~D24を使ってアクセスされます。

040turboのバスサイジング

X68030はX68000の周辺デバイスを流用しているために、16ビットサイズの周辺デバイスに対して働くダイナミックバスサイジング機構が必須です。しかし、8ビットサイズの周辺デバイスはダイナミックバスサイジング機構を持たない68000で使用されていたわけですから、16ビットサイズの周辺デバイスと同じ扱いと思って差し支えありません。

したがって、040turboでは8ビットサイズへのダイナミックバスサイジングは省略し、16ビットサイズのダイナミックバスサイジングのみサポートしています。

以下、個々のロジックについて説明していきましょう。

バスサイクル制御部

この部分は、68040がアクセスするバスサイズと、周辺デバイスからの応答を見てダイナミックバスサイジングが必要かどうかを判定し、バスサイクルを制御する部分です。IC1およびIC2のなかで次のようなロジックで行われています。

IC1のなかのロジック

```
adr_low     = (!EX_ADR1);
next_bus.d  =    AS & s_long & d_word & (DSACK0 $ DSACK1) & adr_low & xta!# next_bus & !xta ;
tsn         = TS # next_bus & !xta;
tsn_wait.d = tsn;
new_tsn     = tsn_wait  & wait_sw
            # tsn       & !wait_sw;
AS.d        = new_tsn   & !xta
            # AS        & xta ÅG
DS.d        = new_tsn   & !xta & read
            # AS        & !xta ;
```

EX_ADR1は、アドレスA1の信号です。

s_longはSIZ0-SIZ1信号をIC3でデコードしたもので、現在のサイクルがロングワードか否かを示しています。

d_wordは、IC1自身のなかで$\overline{\text{DSACK0}}$-$\overline{\text{DSACK1}}$信号をデコードして得られる信号です。

next_busは、68040がロングワードアクセス（s_long）してきて、周辺がワードサイズ（d_word）だったとき、最初のアクセス（adr_low）であれば、IC2からのアクセス終結（xta）を待ってアサートされます。この信号は、ダイナミックバスサイジングの追加サイクルの実行中であることを示します。

このnext_busによってtsnが有効になり、最終的に$\overline{\text{AS}}$、$\overline{\text{DS}}$信号がアサートされて追加のバスアクセスが開始されます。追加のバスアクセスの開始とともに重要なもう1つのポイントは、追加バスアクセスの完了まで68040を待たせるところです。

IC2のなかのロジック

```
s_word   = !s_long
ta_sub.d = AS & (s_word & d_word # s_long & d_word & next_bus) & wait_mask;
TA.d     = ((d_long & wait_mask) #ta_sub)&!AVEC # as_mask & AVEC;

xta.d    = AS & ((d_long # d_word) & wait_mask
           # (BERR # AVEC) & as_mask );
```

$\overline{\text{TA}}$信号は、最初、ロングワードアクセス（s_long）に対し、周辺デバイスがワードサイズ（d_word）であった場合にはアサートされません。したがって、68040はバスサイクルを終結しません。

一方、xtaは、この組み合わせでもアサートされるので、IC1は$\overline{\text{AS}}$信号をネゲートし、X68030側のバスサイクルは完了します。そして、新たにIC1によりダイナミックバスサイジングのためのnext_busがアサートされて次のサイクルが始まります。

next_busがアサートされた状態で周辺デバイスからワードサイズ（d_word）のアクノリッジが返ってきたら、ta_subのほうで認識され、今度は$\overline{\text{TA}}$信号もアサートされるので、最初のサイクルとあわせて68040のロングワードアクセスが完了します。

データバス変換部

この部分は、バスサイジングによる最初のサイクルで得られたデータの保持や、データバスの上位16ビット(D31～D16)と下位16ビット(D15～D0)の間の組み換えを行います。

これは、図4.5のような回路で行われており、8ビットバストランシーバ(74AS245) IC7～12と、8ビットフリップフロップ（74AS374）IC13、IC14から成り立っています。

各々のゲートの制御は、IC3のなかの次のロジックで行っています。

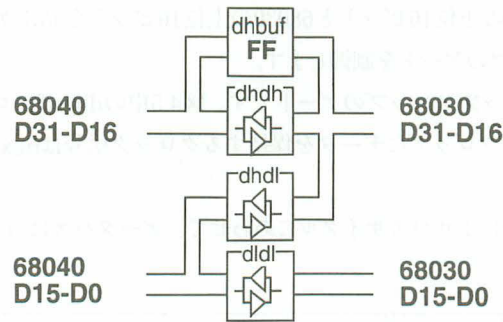



図4.5 バスサイジングのためのバス組み換え回路

```
read        = (RD_WT) ;
write       = (!RD_WT);
adr_low     = (!EX_ADR1);
adr_hi      = (EX_ADR1);
read_adr_hi = read & ( iack # (!iack & adr_hi));
gate        =  TIP & !next_adr;
gate2       =  TIP & next_adr;
tmp_long    = SIZE0 & SIZE1 # !SIZE0 & !SIZE1;
s_long      = tmp_long;
g_dhdh      = gate & (    s_long
                     # !s_long & adr_low );
g_dldl      = gate & (    s_long
                     # !s_long & write & adr_hi
                     # !s_long & read_adr_hi  & !d_word) ;
g_dhdl      = gate & (    !s_long & write & adr_hi
                     # !s_long & read_adr_hi & d_word )
            # gate2;
g_dhbuf     = gate2 & read;
```

iackは、IC4からの割り込みアクノリッジサイクルを示す信号です。68040では、割り込みベクタをつねにデータバスの最下位D7~D0から取り込みますので、16ビットサイズの周辺デバイスがD31~D16に接続されている場合はバスの組み換えをしなければなりません。

$\overline{\text{TIP}}$は、68040がバスアクセスしているとき、アサートされる信号です。68030や他のバスマスタがバスを使用中はバス変換回路をハイインピーダンスにするために、この信号をゲート信号に使っています。

g_dhdhは、68040の上位16ビットと68030の上位16ビットを結ぶゲート信号です。図4.5中のdhdhブロックのゲートを制御します。

g_dldlは、68040の下位16ビットと68030の下位16ビットを結ぶゲート信号です。図4.5

中のdldlブロックのゲートを制御します。

g_dhdlは、68040の下位16ビットと68030の上位16ビットを結ぶゲート信号です。図4.5中のdhdlブロックのゲートを制御します。

g_dhbufは、フリップフロップのゲートです。図4.5中のdhbufブロックのゲートを制御します。フリップフロップにデータを保持するクロック信号はnext_busを使用しています。

これらの制御信号によりバスサイクルにあわせて、データバスは以下のように組み換えられます。

68040アクセス			040turbo	X68030バス		040turboバス変換部ゲート			
SIZ0-1	R/W	A1	next_bus	A1	$\overline{\text{DSACK0-1}}$	g_dhdh	g_dldl	g_dhdl	g_dhbuf
long	R	0	0	0	long	1	1	0	0
long	W	0	0	0	long	1	1	0	0
long	R	0	0	0	word	1	1	0	0
			1	1	word	0	1	0	1
long	W	0	0	0	word	1	1	0	0
			1	1	word	0	0	1	0
word	R	0	0	0	long	1	0	0	0
word	W	0	0	0	long	1	0	1	0
word	R	0	0	0	word	1	0	0	0
word	W	0	0	0	word	0	0	1	0
word	R	1	0	1	long	0	1	0	0
word	W	1	0	1	long	0	1	0	0
word	R	1	0	1	word	0	0	1	0
word	W	1	0	1	word	0	1	1	0

以下、例を挙げて具体的な動作について説明しましょう。

68040からの32ビットリードに対し、周辺デバイスが16ビットサイズの場合を考えます。このときのデータバス組み換え回路の動作を図4.6に、このときのリードサイクルのタイミングを図4.7に示します。

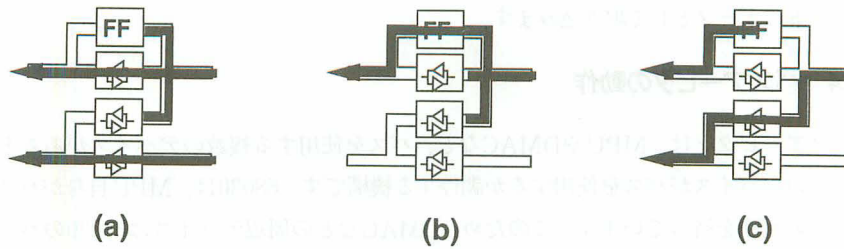



図4.6　バスサイジング時のデータバスの組み換え

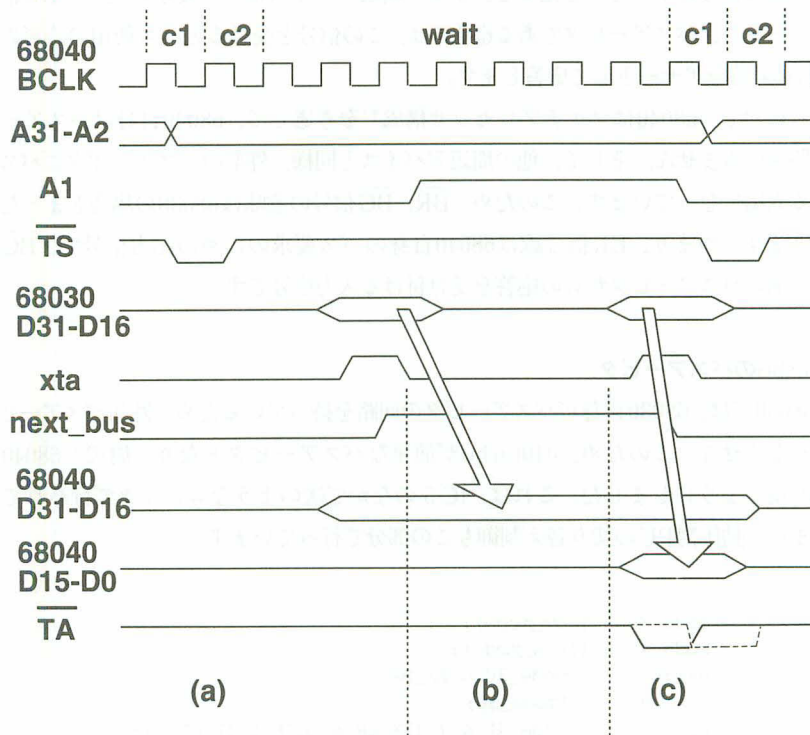



図4.7　バスサイジングによるリードサイクル

(a) 最初は周辺が32ビットか16ビットかわからないので、g_dhdh、g_dldlの両方のゲートが開きます。

(b) 16ビットサイズとわかると、next_busがアサートされ、フリップフロップに最初のデータが保持されるとともに、現在のバスサイクルを終結し、A1＝1として次のバスサイクルを開始します。

(c) 次のバスサイクルでは、g_dhdl、g_dhbufのゲートが開き、データバスの下位に現在のバスサイクルのデータを誘導するとともに、データバスの上位には最初のアクセスで得たデータをフリップフロップから供給します。こうして、68040は32ビッ

トのデータとして取り込みます。

4.1.4 バスアービタの動作

バスアービタとは、MPUやDMACなど、バスを使用する複数のデバイスがあるときに、どのデバイスがバスを使用するか調停する機構です。68030は、MPU自身がバスアービタの回路を持っています。このため、DMACなどの周辺デバイスは68030のバスアービタにバス要求を出し、バス使用権を譲り受けるという形になります。

具体的な信号のやりとりを見ると、まず、周辺デバイスはバス要求のために$\overline{\mathrm{BR}}$信号線をアサートし、バスアービタである68030は、この信号を受けるとバス使用の許可のために$\overline{\mathrm{BG}}$信号線をアサートして応答します。

これに対し、68040はマルチプロセッサ構成*1を考慮して、68040自身はバスアービタの回路を持ちません。そして、他の周辺デバイスと同様、外部のバスアービタにバス要求をする立場になっています。このため、$\overline{\mathrm{BR}}$、$\overline{\mathrm{BG}}$信号の意味は68030の場合とまったく逆になります。つまり、$\overline{\mathrm{BR}}$信号線は68040自身のバス要求のための出力信号で、$\overline{\mathrm{BG}}$信号線は外部のバスアービタからの応答を受け付ける入力信号です。

040turboのバスアービタ

X68030では、68030自身がバスアービタの回路を持っているため、外部バスアービタ*2は存在しません。このため、040turboが簡単なバスアービタとなり、周辺と68040の間を取り持つようにしました。これは、IC5のなかで次のようなロジックで行われています。また、動作MPUの切り替え制御もこの部分で行っています。

```
mode_30 = (now_mode);
mode_40 = (!now_mode);
BR_30 =       mode_30 & EX_BR
   #          !mode_30;
BG_40 =       mode_40 & ( !EX_BR # LOCK & !LOCKE );
bg_wait.d = !BG_40 & !TIP;
EX_BG =       mode_40 & ( !BG_40 & !TIP & bg_wait )
          #   mode_30 &  BG_30;
```

now_modeは、現在の動作MPUが68030か68040かという状態を保持しています。

BR_30は68030へのバス要求信号です。68030モードではEX_BRによる外部からのバ

＊1
バスアービタは、調停という役目上、システムに複数あっては困りますので、68030のようにバスアービタを兼ねているMPUは、マルチプロセッサ構成にするのに都合が悪いのです。

＊2
68040動作時もバスアービタとして68030を使うという手段もありますが、68040のBR信号がスリーステートでないことや、つねにバス要求を出し続けるなど、他の周辺デバイスとは異なる動作をするため、単純にはいきません。

ス要求をそのまま渡していますが、68040モードではつねにバス要求を出すようになっています。この状態では、68030はバスを使えないため、事実上、止まった状態になります。

BG_40は68040へのバス使用許可信号です。EX_BRによる外部からのバス要求がない間はつねにアサートされ、68040にバス使用が許可された状態になっています。この状態は、「暗黙の使用権」と呼ばれており、68040が実際にバスを必要とするとき、すでにバスを使用することが許可されているので、バス使用権確保のためオーバーヘッドすることなく、バスアクセスができます。

EX_BRによるDMACなどの外部デバイスからのバス要求が発生した場合、68040が$\overline{\mathrm{LOCK}}$によりバスをロックしてアクセス[*1]しているときは、その終了を示す$\overline{\mathrm{LOCKE}}$を待ち、ロック状態でなければBG_40をネゲートします。これで68040はバス使用権を失いますから、周辺デバイスがバスを使用できるようになります。また、68030モードの場合は、つねにこのBG_40信号をネゲート状態にしておくことで68040を止めています。

EX_BGは、EX_BR信号でバスを要求してきた外部デバイスに対するバス使用の許可信号です。68030モードのときは68030のバス使用許可信号であるBG_30をそのままつないでいます。68040モードのときは68040へのバス使用許可信号をネゲートして68040のバス使用権を放棄させた後、EX_BGをアサートして、バス使用の許可を出しています。

なお、EX_BGのmode_40のほうは（!BG_40 & !TIP & bg_wait）となっていますが、これは、68040がバス使用許可信号をネゲートしてもすぐにはバスを手放さずバスアクセスしてくることがあるので、1クロックのウェイトを入れるためです。

4.1.5　転送属性信号

68030では、アクセスするアドレス空間の種類をファンクションコードとして、次のようなメモリ空間が定義されており、FC2～FC0の3本の信号線により、どの空間をアクセスしているかが示されるようになっています。

＊1　68030のリードモディファイライトサイクルやバースト転送の場合にアサートされます。

FC2	FC1	FC0	アドレス空間
0	0	0	未定義
0	0	1	ユーザーデータ空間
0	1	0	ユーザープログラム空間
0	1	1	未定義
1	0	0	未定義
1	0	1	スーパーバイザデータ空間
1	1	0	スーパーバイザプログラム空間
1	1	1	CPU空間

68040でもアドレス空間の概念は存在していますが、FC2〜FC0の信号線がなくなり、よりハードウェア寄りの情報を示す「転送属性信号」と呼ばれるTT1〜TT0、TM2〜TM0の5本の信号線が追加されています。TT1〜TT0の意味は次のとおりです。

TT1	TT0	転送タイプ
0	0	ノーマルアクセス
0	1	move16アクセス
1	0	Alternate Logic Function Code Access
1	1	アクノリッジアクセス

また、ノーマルアクセスとmove16アクセスのとき、TM2〜TM0は次の意味となります。

TM2	TM1	TM0	転送モディファイヤ
0	0	0	データキャッシュプッシュアクセス
0	0	1	ユーザーデータアクセス
0	1	0	ユーザーコードアクセス
0	1	1	MMUテーブルサーチデータアクセス
1	0	0	MMUテーブルサーチコードアクセス
1	0	1	スーパーバイザデータアクセス
1	1	0	スーパーバイザコードアクセス
1	1	1	予約

TT1、TT0のアクノリッジアクセスのときは、TM2~TM0には割り込みレベルが出力されます。

040turboで、これらの信号からファンクションコードを作り出して、ハードウェアの差を吸収しています。

ファンクションコード

X68030ではFC2~FC0を見てスーパーバイザ空間の保護を行っているので、68040の動作時もファンクションコードが必要となります。040turboでは転送属性信号からファンクションコードを作り出しています。これは、IC4のなかの次のロジックで行われています。

```
mode_iack = ( TT0 & TT1 );

FIELD   tm040 = [TM2..0];
FIELD   fc030 = [TmFc2..0];

TABLE  tm040 => fc030{
       'b'000 => 'b'101;
       'b'001 => 'b'001;
       'b'010 => 'b'010;
       'b'011 => 'b'101;
       'b'100 => 'b'110;
       'b'101 => 'b'101;
       'b'110 => 'b'110;
       'b'111 => 'b'111;
    }

       FC2    = mode_iack # TmFc2;
       FC1    = mode_iack # TmFc1;
       FC0    = mode_iack # TmFc0;
```

TABLE tm040 => fc030は真理値表形式の信号の対応を定義しています。

データキャッシュプッシュアクセス(000)、およびMMUテーブルサーチ(011, 100)は、68040で新設された状態で、68030には対応するものがないので、互換性のためスーパーバイザデータ空間(101)とコード空間(110)に割り付けています。

また、TT1、TT0で示される割り込みアクノリッジサイクルは、68030ではCPU空間で行われていたものですから、強制的にFC2~FC0をCPU空間(111)にしています。

割り込みレベルの表示

68030では割り込みアクノリッジサイクルにおいて、受け付けた割り込みレベルをアドレス線のA4~A1で表示していましたが、68040ではTM2~TM0で表示するようになっています。これは、IC4のなかの次のロジックで行われています。

```
mode_iack = ( TT0 & TT1 );

a3=s_long;
a2=next_adr;
a1=ADR1;
a0=ADR0;

FIELD adr040 = [a3..a0];
FIELD adr030 = [adrl,adr0];

TABLE adr040 => adr030 {
    'b'0000 => 'b'00; /* not long (!s_long) */
    'b'0001 => 'b'01;
    'b'0010 => 'b'10;
    'b'0011 => 'b'11;
    'b'0100 => 'b'00;
    'b'0101 => 'b'01;
    'b'0110 => 'b'10;
    'b'0111 => 'b'11;
    'b'1000 => 'b'00; /* normal long (s_long & !next_bus) */
    'b'1001 => 'b'00;
    'b'1010 => 'b'00;
    'b'1011 => 'b'00;
    'b'1100 => 'b'10; /* next long (s_long & next_bus) */
    'b'1101 => 'b'10;
    'b'1110 => 'b'10;
    'b'1111 => 'b'10;
  }

EX_ADR3 = mode_iack & TM2 # !mode_iack & ADR3;
EX_ADR2 = mode_iack & TM1 # !mode_iack & ADR2;
EX_ADR1 = mode_iack & TM0 # !mode_iack & adr1;
EX_ADR0 = mode_iack & 'b'1 # !mode_iack & adr0;
```

基本的には、X68030側のアドレス線A4～A1に対し、割り込みアクノリッジサイクルの場合はTM2～TM0の内容を出力し、通常のアクセスでは68040のアドレスの内容を出力します。

TABLE adr040 ＝> adr030の定義は一見複雑ですが、これはダイナミックバスサイジングによる追加のアクセスで、次のワードデータをアクセスするためにアドレスA1を0から強制的に1にするロジックが含まれているからです。

4.1.6　その他の信号線の意味

68040では、今まで説明してきたようなバス動作の違いに加え、細かい信号線の意味づけや使われ方が変更されています。ここでは、それらの違いと040turboによる対応について説明します。

サイズ信号

68030では、SIZ0~SIZ1の2ビットの組み合わせでロングワード転送、バイト転送、ワード転送、3バイト転送を示していたのに対し、68040では、ロングワード転送、バイト転送、ワード転送、ライン転送*1を示すようになっています。

X68030はバースト転送をサポートしていないので、68040に対してTBI信号を返してライン転送は4つのロングワード転送に分割させています。このため、ライン転送はロングワード転送と同じ扱いでかまいません。

このサイズ信号の変換は、IC3のなかの次のロジックで行われています。

```
adr_hi = (EX_ADR1);
s0 = SIZE0;
s1 = SIZE1;
s2 = adr_hi;

FIELD size30 = [s2..s0];
FIELD size40 = [EX_SIZE1,EX_SIZE0];

TABLE size30 => size40 {
      'b'000  => 'b'00; /* long */
      'b'001  => 'b'01; /* byte */
      'b'010  => 'b'10; /* word */
      'b'011  => 'b'00; /* line */
      'b'100  => 'b'10; /* long->word */
      'b'101  => 'b'01; /* byte */
      'b'110  => 'b'10; /* word */
      'b'111  => 'b'10; /* line->word */
```

SIZ0~SIZ1が68040側のサイズ信号で、EX_SIZ0~EX_SIZ1がX68030側に出力するサイズ信号です。

EX_ADR1はIC4から出力されているアドレスA1信号で、これを見てダイナミックバスサイジングで挿入されたサイクルについてはワードサイズに変更しています。

リセット信号

68030では、$\overline{\text{RESET}}$信号は入出力の双方向信号です。一方、68040では、68040に対しリセットをかける$\overline{\text{RSTI}}$と、68040から周辺デバイスに対しリセットをかける$\overline{\text{RSTO}}$の信号に分かれています。これらの変換は、IC5のなかで次のロジックで行われています。

```
EX_RST     = 'b'1;
EX_RST.oe  = RSTO;

rsto_msk   = RSTO # !TIP & rsto_msk & !EX_BG;

RSTI       = EX_RST & ( !RSTO & !rsto_msk );
```

＊1
キャッシュフィルやキャッシュプッシュ、move16命令で使用される4ロングワードの連続転送サイクルを示します。

“.oe”がついている信号は、その信号をハイインピーダンスにするかどうかの制御です。これにより、$\overline{\mathrm{RSTO}}$信号がアサートされるとEX_RSTがアサートされ、$\overline{\mathrm{RSTO}}$信号がネゲートされるとEX_RSTがハイインピーダンスになります。

$\overline{\mathrm{RSTI}}$は外部のEX_RSTを入力としていますが、68040自身が$\overline{\mathrm{RSTO}}$で外部に対しEX_RSTをドライブしているときに自分にリセット信号が戻ってこないようにマスク[*1]しています。

68030にのみ存在する出力信号

次の信号は、040turboでは何も対処していません。

$\overline{\mathrm{ECS}}$、$\overline{\mathrm{OCS}}$、$\overline{\mathrm{RMC}}$、$\overline{\mathrm{DBEN}}$、$\overline{\mathrm{STERM}}$、$\overline{\mathrm{CBREQ}}$、$\overline{\mathrm{CBACK}}$、$\overline{\mathrm{REFILL}}$、$\overline{\mathrm{IPEND}}$、$\overline{\mathrm{STATUS}}$、$\overline{\mathrm{HALT}}$

68040にのみ存在する入力信号

次の信号は、それぞれ適当に処理しています。

$\overline{\mathrm{TBI}}$、TCK、$\overline{\mathrm{TRST}}$、SC1-0 ―― GNDにプルダウン
TMS、TDI、PST3-0、TLN1-0、UPA1-0、TDO、MI ―― オープン

68030と68040で名前が異なるもの

040turboでは、次のように接続しています（厳密な意味では、多少動作が違うかもしれません）。

68030		68040
$\overline{\mathrm{CIOUT}}$	――	$\overline{\mathrm{CIOUT}}$
$\overline{\mathrm{CIIN}}$	――	$\overline{\mathrm{TCI}}$
$\overline{\mathrm{CDIS}}$	――	$\overline{\mathrm{CDIS}}$

その他

040turboでは、DLE（Data Latch Enable）モードを利用するため、68040の$\overline{\mathrm{MDIS}}$信号をリセット中アサートしなければなりません。本来の意味からすると、$\overline{\mathrm{MDIS}}$信号は

＊1
68040のユーザーズマニュアルには、68040の$\overline{\mathrm{RSTO}}$をオープンコレクタを介して外部リセットおよび$\overline{\mathrm{RSTI}}$に接続するような例が書いてあります。しかし、68040からreset命令で$\overline{\mathrm{RSTO}}$をアサートしたとたん、68040自身がリセットされてしまったので、このようにマスクするロジックとしました。

68030の$\overline{\mathrm{MMUDIS}}$信号につなぐべきですが、$\overline{\mathrm{MMUDIS}}$信号はX68030中で$\overline{\mathrm{CDIS}}$信号などといっしょにプルアップされているようです。このため、040turboでは、$\overline{\mathrm{MMUDIS}}$信号との接続をあきらめ、$\overline{\mathrm{MDIS}}$信号を$\overline{\mathrm{RSTI}}$信号につないでDLEモードになるようにしています。

4.2　68040特有の動作

4.2.1　キャッシュプッシュアクセス

転送属性信号のところで少し触れましたが、キャッシュプッシュアクセスは68040で追加されたコピーバックモードに対応したものです。コピーバックモードでは、データ書き込み命令を実行しても、キャッシュに書き込んで、そのキャッシュエントリのダーティビットを立てておくだけで、すぐにはメモリ書き込みを行いません。キャッシュエントリが変わるときや、明示的にキャッシュプッシュ命令を実行したとき、はじめてキャッシュからメモリに書き戻されます。これは、時間のかかるメモリアクセスをともなわないので頻繁にデータが更新される場合、特に有効な手段です。

しかし、DMAなどがバスを使うときにキャッシュプッシュされていないとメモリ上の古いデータのまま[*1]で処理されてしまいます。

また、データキャッシュと命令キャッシュが独立しているため、自己書き換えプログラム[*2]では問題が発生します。

さらに、キャッシュプッシュアクセスは、ユーザーモードで書き込まれたデータか、スーパーバイザモードで書き込まれたデータかの区別がないので、X68030のファンクションコードによるメモリ保護がきかなくなります。68040クラスであれば、アドレス空間といったものではなく、MMUを使った保護をすべきでしょう。

パッチプログラム040SYSpatch.xでは、MMUを使ってHuman68kの領域をスーパーバイザ領域に設定することで、ユーザーモードで破壊できないように保護しています。

＊1
キャッシュをプッシュしないでDMAなどのほかのバスマスタを動作させる手法としては「バススヌーピング」というものがあります。これは、ほかのバスマスタの動作を68040が監視し、未プッシュのデータに該当するアクセスがあった場合にメモリにかわって68040が内部キャッシュのデータを返すという方法です。しかし、これをサポートするためには一時的にメモリをインヒビットする機能が必要となります。もちろん、X68030にはありませんので、040turboではサポートしていません。

＊2
68030でも、命令キャッシュに載っている部分がメモリ上で書き換えられたときに破綻します。これは、命令キャッシュの内容が古いままなので、命令キャッシュをクリアしなければなりません。68040のコピーバックキャッシュでは、さらにメモリ上で書き換えられたはずの部分が、実はまだメモリ上に書き戻されていないという状況も発生します。この場合は、命令キャッシュのクリアの前にデータキャッシュのプッシュも必要になります。

4.2.2　バースト転送

68030ではシングルアクセスが基本で、ソフトウェアでバースト転送の設定を行ったとき、$\overline{\text{CBREQ}}$信号をアサートしバースト転送を試みます。しかし、X68030はバースト転送のハードウェアをサポートしないので、$\overline{\text{CBACK}}$信号はネゲートされており、バースト転送はアボートします。

これに対し、68040ではバースト転送が基本で、4ロングワードにまとめたラインの転送を試みます。バースト転送が可能か否かは$\overline{\text{TBI}}$信号で指定します。$\overline{\text{TBI}}$信号をアサートして、バースト転送を禁止された場合、68040はバースト転送のかわりに4つのロングワード転送を実行します。いずれにしても、68040では命令をどんどん先読みしていくので、実装メモリぎりぎりの部分にプログラムを置くと、先読みでバスエラー*1になる可能性があります。

また、データキャッシュをオンにしておくと、いつでもライン単位すなわち4ロングワード単位でアクセスをしようとしますので、I/Oアクセスのときに指定していないアドレスもアクセスされる恐れがあります。

パッチプログラム040SYSpatch.xでは、MMUを使ってI/O領域やVRAM、スプライトメモリなどをキャッシュ禁止でアクセスされるように設定することで、これを回避しています。

4.2.3　バスアクセスのノン・シリアライズドアクセス

68040のバスアクセスは、必ずしも命令コードの順に行われるとはかぎりません。後ろに置かれた命令が先に実行される*2ことが起こりえます。I/Oアクセスの順番が変わる可能性があって問題になるかもしれません。

68040は、ソフトウェアによりデータアクセスを命令の順番どおりに行うシリアライズドアクセス(キャッシュオフになる)にすることもできます。040SYSpatch.xでは、MMUを使ってI/O領域をキャッシュ禁止のシリアライズドアクセスされるように設定しています。ちなみに、68040はキャッシュオフのノン・シリアライズドアクセスがデフォルト動作です。

＊1
68000とは違って、68040では先読みでバスエラーが起こっても、その命令が実際に実行されるところにくるまでエラーは保留されます。もし、その前に分岐が起こり、実行されなければ、エラーにはなりません。

＊2
movep命令の実行時に発生しました。しかし、ほかではめったに起こらないようです。

4.2.4　その他

このほかにも、68040は68030より高速化しているため、ソフトウェアループでタイミングをとっている場合などは異常が発生する恐れがあります。これらについてはハードウェアでは対処できないので、ソフトウェアごとにパッチを当てることで対処することになります。

第5章
040turbo対応ソフトウェア説明

68000からすでに32ビット指向の設計であるため、ソフトウェアの互換性は高いといえます。また、68000と68030の差異についてはX68030のシステムソフトウェアである程度吸収されているので、キャッシュオフでは68040の動作に関してはほぼ互換性があるといえるでしょう。実際、何のパッチを当てなくても動きます。*1

キャッシュオンでは68030と68040の違いがストレートに出ていますが、040SYSpatch.xでほとんどの問題は押さえ込んでいます。Human68kを使っている分にはほとんど68030と68040の違いを気にする必要はありません。

しかし、68040と68030のどの点が異なり、040turboではどう対処しているかについて理解しておくのは意味のあることです。68040を使いこなせるようになって040turboプロジェクトに積極的に参加してください。

5.1　68040と68030との相違

モトローラは、ユーザーモード互換をいうわりに、スーパーバイザモードでの互換性については無頓着です。ハードウェアほど互換性がないわけではありませんが、それでも次のような点に互換性のない部分があります。

1. キャッシュ
2. 浮動小数点演算プロセッサ
3. MMU

以下、それぞれの相違点とその対処のしかたについて説明します。

5.1.1　キャッシュ

68030のキャッシュ制御は、キャッシュ制御レジスタCACRおよびキャッシュアドレスレジスタCAARの2つの制御レジスタにより行うようになっています。

68040では、同じキャッシュ制御レジスタCACRでも、制御ビットの位置、働きが異な

＊1　Human68kのみならず、OS 9-X68030も動きました。

っています。また、キャッシュアドレスレジスタは存在せず、かわりにキャッシュメンテナンスのための命令が新設されました。

キャッシュ制御は040turboのパッチの最重要ポイントですから、68030および68040のそれぞれのキャッシュ制御方法について、詳細に説明します。

■68030のキャッシュ制御

68030のキャッシュ制御は、キャッシュの制御をするCACRと、その際のアドレスを指定するCAARから[*1]成り立っています。

レジスタフォーマットは次のとおりです。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	WA	DBE	CD	CED	FD	ED	0	0	0	IBE	CI	CEI	FI	EI

CACR

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
—																								cache-index						—	

CAAR

●68030のキャッシュ関連レジスタ

各ビットの意味は、次のとおりです。

WA：ライトアロケート

ライトアロケーションモード(WA＝1)かノーライトアロケーションモード(WA＝0)を選択します。ライトアロケーションモードではライトサイクルでキャッシュが更新されますが、ノーライトアロケーションモードではミスヒット（ライトアドレスがキャッシュタグと不一致）した場合、キャッシュは更新されません。

DBE：データキャッシュのバースト転送イネーブル

このビットをセットすると、データキャッシュの充塡のためにバースト転送が用いられるようになります。

CD：データキャッシュのクリア

このビットをセットすると、データキャッシュの全エントリがクリアされます。

CED：データキャッシュの特定エントリのクリア

このビットをセットすると、CAARで指定したデータキャッシュのエントリがクリアされます。

＊1
68EC030では、さらにキャッシュ範囲を指定するレジスタが追加されていますが、Human68kでは使用されていないようですので無視します。

FD：データキャッシュのフリーズ

このビットをセットすると、データキャッシュがフリーズされます。ただし、ライトサイクルでキャッシュヒットした場合は新しいデータに更新されます。

ED：データキャッシュのイネーブル

このビットをセットすると、データキャッシュがイネーブルされます。なお、データキャッシュをディセーブルしてもキャッシュはクリアされません。再度イネーブルにすると、以前の有効エントリがそのまま有効になります。

IBE：命令キャッシュのバースト転送イネーブル

このビットをセットすると、命令キャッシュの充填のためにバースト転送が用いられるようになります。

CI：命令キャッシュのクリア

このビットをセットすると、命令キャッシュの全エントリがクリアされます。

CEI：命令キャッシュの特定エントリのクリア

このビットをセットすると、CAARで指定した命令キャッシュのエントリがクリアされます。

FI：命令キャッシュのフリーズ

このビットをセットすると、命令キャッシュが凍結フリーズされます。

EI：命令キャッシュイネーブル

このビットをセットすると、命令キャッシュがイネーブルされます。なお、命令キャッシュをディセーブルしてもキャッシュはクリアされません。再度イネーブルにすると、以前の有効エントリがそのまま有効になります。

cache-index：インデックス

CED、CEIをセットして特定エントリをクリアするときのキャッシュエントリを指定します。

CACR、CAARなどの制御レジスタへのアクセスは、movec命令で行います。movec命令のフォーマットは次のとおりです。

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	1	1	1	0	0	1	1	1	1	0	1	dr
A/D	register			control-register											

16進で、$4E7Axxxxもしくは$4E7Bxxxxになります。

dr、A/D、register、control-registerの各フィールドの意味は次のとおりです。なお、制御レジスタのアクセスは、データレジスタもしくはアドレスレジスタとの間でしか行えません。

dr	転送方向
0	制御レジスタの内容を汎用レジスタにセット
1	汎用レジスタの内容を制御レジスタにセット

register	A/D＝0	A/D＝1
000	D 0	A 0
001	D 1	A 1
010	D 2	A 2
011	D 3	A 3
100	D 4	A 4
101	D 5	A 5
110	D 6	A 6
111	D 7	A 7

制御レジスタには以下のようなものがあります。CACRは002（16進）、CAARは802（16進）です。

control-register（16進）	制御レジスタ名
000	SFC
001	DFC
002	CACR
003	TC
004	ITT 0
005	ITT 1
006	DTT 0
007	DTT 1
800	USP
801	VBR
802	CAAR
803	MSP
804	ISP
805	MMUSR
806	URP
807	SRP

例）

・CACRの値をD0に読み出す。
　アセンブラ表記　：movec CACR,D0
　マシンコード表記：$4E7A0002

・A 2 の内容をCAARに設定する
　アセンブラ表記　：movec Az,CAAR
　マシンコード表記：$4E7BA802

■68040のキャッシュ制御

68040のキャッシュ制御レジスタCACRでは、単に命令キャッシュ、データキャッシュのオン・オフを指定するだけで、キャッシュクリアなどは68040で新設されたキャッシュ

メンテナンス命令で行います。また、キャッシュのフリーズ機能はなくなっています。

レジスタのフォーマットは次のとおりです。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	IE	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

CACR

●68040のキャッシュ制御レジスタ

各ビットおよび命令の意味は、次のとおりです。

DE：データキャッシュのイネーブル

このビットをセットすると、データキャッシュがイネーブルされます。なお、データキャッシュをディセーブルしてもキャッシュはクリアされません。再度イネーブルにすると、以前の有効エントリがそのまま有効になります。

IE：命令キャッシュイネーブル

このビットをセットすると、命令キャッシュがイネーブルされます。なお、命令キャッシュをディセーブルしてもキャッシュはクリアされません。再度イネーブルにすると、以前の有効エントリがそのまま有効になります。

CACRレジスタ自体は、68030と同様、movec命令が新設され、これを使ってキャッシュクリアなどを使ってアクセスします。レジスタ番号も同じ002（16進）です。

●68040のキャッシュメンテナンス命令

68030ではCACRレジスタのビットを立ててキャッシュクリアの処理を行っていましたが、68040ではCINV、CPUSHというキャッシュメンテナンス命令が新設され、これを使ってキャッシュクリアなどを行うようになりました。

・CINV：キャッシュの無効化命令

キャッシュエントリを強制的に無効（インバリッド）にします。

・CPUSH：キャッシュの書き戻しと無効化命令

コピーバックモードでメモリに書き戻されていないデータがキャッシュエントリにある場合、それをメモリに書き戻した後、無効にします。

どちらの命令も、作用するキャッシュエントリの範囲により、次の3つに分けられます。

CINVL caches, (An)　アドレスレジスタAnで指定されるキャッシュラインを無効化します。cachesは、命令・データ・両方のどれかを指定します。

CINVP caches,(An)　アドレスレジスタAnで指定されるページ内に含まれるキャッシュエントリを無効化します。cachesは、命令・データ・両方のどれかを指定します。

CINVA caches　cachesで指定されるキャッシュのエントリをすべて無効化します。

CPUSHも同様です。

CPUSHL caches,(An)
CPUSHP caches,(An)
CPUSHA caches

命令フォーマットは次のとおりです。

15	14	13	12	11	10	9	8	7 6	5	4 3	2 1 0
1	1	1	1	0	1	0	0	caches	0	scope	register

CINV

15	14	13	12	11	10	9	8	7 6	5	4 3	2 1 0
1	1	1	1	0	1	0	0	caches	1	scope	register

CPUSH

caches、scope、registerの各フィールドの意味は次のとおりです。

caches	対象キャッシュ
00	——
01	DC：データキャッシュ
10	IC：命令キャッシュ
11	IC/DC：データ／命令キャッシュ

scope	対象範囲
00	無効
01	CINVL
10	CINVP
11	CINVA

register	対象レジスタ
000	A 0
001	A 1
010	A 2
011	A 3
100	A 4
101	A 5
110	A 6
111	A 7

例）
・全キャッシュをクリアする
　アセンブラ表記　：CINVA IC/DC
　マシンコード表記：＄F4D8
・A5の指すラインのデータキャッシュだけをプッシュする
　アセンブラ表記　：CPUSH DC,(A5)
　マシンコード表記：$F46D

これらの違いにより、68030と68040のキャッシュ関連の処理がどう変わってくるか、具体的に説明します。

■68030と68040のキャッシュのオン・オフの違い

68030では、68030のCACRレジスタのEI、EDビットを操作してキャッシュのオン・オフを行います。X68030のROMでは、WAビットも同時に設定しています。設定値は、次のとおりです。

	データキャッシュ	命令キャッシュ	設定値(16進)
68030	オフ	オフ	0000
	オフ	オン	0001
	オン	オフ	2100
	オン	オン	2101

68040では、CACRのIE、DEビットでキャッシュのオン・オフを行います。040turboのパッチでは、次の値になるようにパッチしています。

	データキャッシュ	命令キャッシュ	設定値(16進)
68040	オフ	オフ	00000000
	オフ	オン	00008000
	オン	オフ	80000000
	オン	オン	80008000

注：ライトアロケートビットについて

68030では、データキャッシュのオンでライトアロケートビットを設定しています。これは、以下の場合に不具合が生じるからです。

1. MMUで複数の論理アドレスを同一物理アドレスにマッピングした場合
2. 同一物理アドレスをスーパーバイザモードとユーザーモードの両方でアクセスする場合

68040にはライトアロケートビットの設定が存在しませんが、キャッシュはMMUによりアドレスに変換された後の物理アドレスでキャッシュされるので、1.の問題はありません。また、スーパーバイザモードとユーザーモードでのキャッシュの区別は特に設定しないかぎり行われないので、2.の問題も発生しません。

■68030と68040のキャッシュのクリアの違い

68030では、CACRのCD、CED、CI、CEIビットを操作してキャッシュのクリアを行います。X68030のROMやHuman68kでは、次の値をCACRの現在値にORしたり、ビットセット命令を使って対応ビットを立ててCACRに書き込み、キャッシュクリアを行っています。

	データキャッシュ	命令キャッシュ	設定値(16進)
68030	――	――	0000
	――	クリア	0008
	クリア	――	0800
	クリア	クリア	0808

68040では、キャッシュメンテナンスのためのCINVもしくはCPUSH命令で行います。040turboのパッチでは、68030のキャッシュクリアのためのCACR操作の一連のコードを、次の命令になるようにパッチしています。

	データキャッシュ	命令キャッシュ	命令
68040	――	――	――
	――	クリア	CINVA IC
	クリア	――	CINVA DC
	クリア	クリア	CINVA IC/DC

なお、68040のキャッシュエントリはリセットではクリアされませんので、最初にキャッシュオンにする前に「CINVA IC/DC」を実行し、全キャッシュエントリをクリアしておかなければなりません。

■キャッシュの指定エントリのクリア

68030では、CAARレジスタで特定のキャッシュエントリを指定し、CACRレジスタのCED、CEIビットにより部分的にクリアする機能を持っています。68040では、CINVL命令、およびCINVP命令により同等の処理をすることが可能です。

しかし、Human68kでは使用されていないので、040turboでは対処していません。

■キャッシュのフリーズ

68030では、CACRレジスタのFD、FIビットによりキャッシュをフリーズする機能を持っています。68040では、キャッシュのフリーズ機能はありません。

これもHuman68kでは使用されていないので、040turboでは対処していません。

■ライトスルーモードとコピーバックキャッシュモード

68030のキャッシュは、データ書き込み時にキャッシュの更新とともに実際の書き込み動作も行われます。68040のキャッシュも、デフォルトではこの動作となり、これを特に「ライトスルーモード」と呼んでいます。これに対し、データ書き込み時はキャッシュのみ更新し、実際の書き込み動作は必要に応じて行うコピーバックモードと呼ばれる動作モードも持っています。

コピーバックモードの指定は、透過変換レジスタかMMUのページディスクリプタで行います。透過変換レジスタのフォーマットは次のとおりです。

<table>
<tr><td>31</td><td>30</td><td>29</td><td>28</td><td>27</td><td>26</td><td>25</td><td>24</td><td>23</td><td>22</td><td>21</td><td>20</td><td>19</td><td>18</td><td>17</td><td>16</td><td>15</td><td>14</td><td>13</td><td>12</td><td>11</td><td>10</td><td>9</td><td>8</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td colspan="8">論理アドレスベース</td><td colspan="8">論理アドレスマスク</td><td>E</td><td colspan="2">S</td><td>0</td><td>0</td><td>0</td><td>U1</td><td>U0</td><td>0</td><td colspan="2">CM</td><td>0</td><td>0</td><td>W</td><td>0</td><td>0</td></tr>
</table>

68040の透過変換レジスタ（ITT 0、ITT 1、DTT 0、DTT 1）

各ビットの意味は、次のとおりです。

論理アドレスベース　アクセスするアドレスのA31～A24と比較され、この８ビットと一致する場合はMMUの変換を受けずに直接アドレスバスに出力されます。

論理アドレスマスク　論理アドレスベースと比較するとき、１がセットされているビット位置に対応するアドレスについては無視されます。

E：イネーブル　このビットをセットすると、トランスペアレントレジスタの設定が有効になります。

S：スーパーバイザ／ユーザーモード　このビットは、アドレス空間を識別するのに使用します。

S	アドレス空間
00	ユーザーモードアクセスのときのみ有効
01	スーパーバイザモードアクセスのときのみ有効
1x	スーパーバイザ／ユーザーモードどちらでも有効

U0、U1：ユーザーページ属性　トランスペアレントレジスタによるアクセスのとき、68040のバス上にあるUPA0、UPA1線に本ビットの内容が出力されます。２次キャッシュの制御などの用途に使用しますが、040turboでは使用していません。

CM：キャッシュモード　このビットで、次のようにキャッシュのモードを指定します。

CM	キャッシュモード
00	ライトスルーキャッシュオン
01	コピーバックキャッシュオン
10	キャッシュオフ（シリアライズドアクセス）
11	キャッシュオフ（ノンシリアライズドアクセス）

W：書き込み保護　このビットがセットされている場合、書き込もうとするとエラーになります。

68040では、この透過制御レジスタが、命令用に２セット（ITT０、ITT１）、データ用に２セット（DTT０、DTT１）あります。

たとえば､DTT０に00000001_00000000_10000000_00100000(２進)とセットした場合、

論理アドレスベース ── 000000001
E ── １（透過レジスタ有効）
S ── 00（ユーザーモードのみ有効）
CM ── 01（コピーバックキャッシュ有効）

となりますから、$01000000～$01FFFFFF（16進）の間をユーザーモードでデータアクセスするときはコピーバックキャッシュが有効になります。X68030では、アドレスバスの上位８ビットがデコードされていないので、メインメモリがコピーバックモードでアクセスされることになります。MMUの場合は、ページディスクリプタでキャッシュモードで指定します。040SYSpatch.xは、この機能を使ってページ単位にキャッシュモードを指定できるようにしています。

5.1.2　メモリ管理ユニット

68040の持つメモリ管理機能は、68030のオンチップMMU*1のサブセットとなっており、使用できない機能があります。

今のところ、MMUの違いで問題が出るのはX68030のIPLだけで、68030が搭載されている場合、「On-Chip MMU」の表示をするためにMMU命令を使ってチェックしていますが、MMU機能の差異のため、68040のMMUは認識されません。実害はないので、見ための表示をごまかすだけしか行っていません。

＊１
68EC030にはありません。

040SYSpatch.xでは、スーパーバイザ領域やキャッシュモードの設定を行うために独自にMMU機能を使用しています。

現時点では、Human68kからMMU機能を使用するソフトウェアが存在しないため問題ありませんが、今後、システムがMMU機能を使用するようになった場合は、MMU機能の差異のほか、040SYSpatch.xが独自に使っている機能との整合をとる必要が出てくるでしょう。

5.1.3　浮動小数点演算ユニット

68040になって浮動小数点演算ユニットはMPU内に内蔵され、コプロ方式の68030＋68882により浮動小数点演算が高速化されました。しかし、この浮動小数点演算ユニットは68882のサブセットであり、次の命令しかサポートされておらず、超越関数などはソフトウェア*1で対応しなければなりません。

ニーモニック	機能
FABS	浮動小数点絶対値
FADD	浮動小数点加算
FBcc	浮動小数点条件分岐
FCMP	浮動小数点比較
FDBcc	デクリメントと浮動小数点条件分岐
FDIV	浮動小数点除算
FMOVE	浮動小数点レジスタ転送
FMOVEM	浮動小数点レジスタの複数転送
FMUL	浮動小数点乗算
FNEG	浮動小数点符号反転
FRESTORE	浮動小数点内部状態リストア
FSAVE	浮動小数点内部状態セーブ
FScc	条件による浮動小数点レジスタの設定
FSQRT	浮動小数点平方根
FSUB	浮動小数点減算
FTRAPcc	浮動小数点条件トラップ
FTST	浮動小数点テスト

この差異のため、X68030のコプロセッサを利用した浮動小数点演算パッケージFLOAT4.xが使用できません。しかし、現状では、すべてソフトウェアで浮動小数点演算を行う演算パッケージFLOAT2.xを使用することができますので、通常のプログラムの使用には問題ありません。また、フリーソフトウェアで68040対応のfloat040.xおよびpfloat.xが用意されています。

*1 モトローラは、それでも68040のほうが68030＋68882より速いといっています。

5.2　040turboパッチプログラム

040turboで用意しているプログラムの使用方法についてはすでに説明しました。ここでは、最も基本となるパッチプログラムである、040SYSpatch.xについて説明します。この内容を理解すれば、他のプログラムを040turboに対応させることは可能でしょう。

040SYSpatch.xは、X68030のROMルーチン、Human68k、SCSIデバイスドライバなどへのパッチや、キャッシュ制御のためのコード追加、MMUの管理など、多岐にわたっていますので、動作が複雑になっています。個々の内容の説明の前に、プログラム自体のおおまかな動作について説明しておきましょう。

1. 引数チェック

・オプションスイッチの引数チェック。

・ROMコピー領域、MMUテーブルのための領域を、$1C00番地*1の内容から必要メモリ容量分だけ引いて$1C00に設定。

2. MPUチェック

MPUが68040かどうかを、「movec tc,d 0」命令*2が未定義命令エラーになるかどうかでチェックしています。

未定義命令のベクタをフックしておき、この命令を実行してみて、正常に実行できれば68040、フックしておいたルーチンに飛んできた場合は68030ということになります。68040でなければ以後の処理はスキップします。

3. コピー領域が正しいか否かのチェック

ROMのパッチをしたコピー領域が残っているかどうかを次の項目に従ってチェックします。

・IOCS-$8F［ROMVER］によるROMのバージョンが正しいか

・コピー領域のマジックナンバーがあっているか

・コピー領域の全体のチェックサムがあっているか

正しくなければ、ROMをコピー領域にコピーしてパッチを当てます。以前のバージョンでは、この後、パッチを当てたIPLで起動しなおすため、RESET時のエントリにジャ

＊1
フリーエリアの最後尾のアドレスを保持しています。この値を修正すれば、Human68kで使われるフリーエリアを変更することができます。

＊2
tcは68040のMMU用の制御レジスタです。ちなみに68030のMMU用の制御レジスタは、pmove命令でアクセスするようになっています。

ンプして2回のリセットが行われていました。今は1回で起動するようになっています。

4. MMUテーブルの作成

68040のスーパーバイザ領域の保護とI/O領域へのキャッシュ禁止のシリアライズドアクセスを設定し、さらにROMのコピー領域を本来ROMがあるアドレスに見せかけるためにMMUを使用します。このためのMMUのテーブルを作成する必要があります。

5. Human68kのパッチ

ROMのパッチの後は、Human68kやSCSIドライバ等にパッチを当てます。

Human68kのバージョンチェックを行ってから、対応箇所をパッチします。

SCSIドライバについては、デバイスドライバのリンクテーブルをたぐっていって対応ルーチンをすべてパッチします。

これらはRAM上にあるので、直接、メモリ上のプログラムコードを書き換えています。

6. 終了

すべての処理が終わったら、終了します。なお、コマンドラインから実行した場合は、単にバージョンを表示するだけです。

次に、個々のパッチ内容について説明していきます。パッチ内容は大きく分けて、

・キャッシュ制御関連
・MMUの管理

になります。

なお、プログラムが頻繁にバージョンアップしているので、プログラムのドキュメントおよびソースのほうも確認しておいてください（'94年4月14日現在の最新バージョンはver2.51です）。

5.2.1 キャッシュ制御関連

040SYSpatch.xの大半のパッチは、68030と68040のキャッシュ制御の違いに関するものです。

次の箇所にパッチを当てています。

■IPLのMPUクロック判定ルーチンに入る前のキャッシュ制御

クロック判定のためのループをキャッシュオンで回るための、次のコードが入っています。

```
　　本来のルーチン　　　　　　　　68040のパッチ
00FF014A move.w #$2101,D0    → move.l #$00000000,D0
00FF014E movec  d0,CACR      → movec  d0,CACR
```

68040では設定値が異なるので、これではキャッシュがオンになりません。設定値を変えてやればいいのですが、キャッシュをオンにすると動作が異常になったことがあるので、パッチ後のルーチンではCACRに$00000000を入れてキャッシュオフにしています。なお、命令キャッシュのみオンなら正しく動く場合もあるので、040SYSpatch.xのオプション"%"により、$00008000をCACRに入れるようにしています。この場合は、命令キャッシュがオンになり、ほぼ正しいクロック値が表示されるようになります。

IOCS内の各ルーチンから呼ばれているキャッシュクリアルーチン

IOCS-ROM内でDMAを使用するルーチン等が、キャッシュクリアのために呼び出している共通ルーチンがあります。

```
　　本来のルーチン　　　　　　　　68040のパッチ
00FF828E movec CACR,d0       → CPUSHA IC/DC
00FF8292 or.w  #$8080,d0
00FF8296 movec d0,CACR
00FF829A and.w #$F7F7,d0
00FF829E movec d0,CACR
```

このルーチンではキャッシュクリアのためにCACRにCD、CIビットをセットしています。

68040では、CPUSHA IC/DCを使うようにパッチしています。

■IOCS-$ACのキャッシュ制御コール

IOCS-$AC［SYS_STAT］は、D1で次のような機能を使用できます。

D1	機能
0	MPU情報の取得
1	キャッシュ状態の取得
2	SRAM設定値にキャッシュモードを設定
3	キャッシュクリア
4	キャッシュモードの設定

MPU情報の取得　MPU情報は、D0レジスタに各種の内容を返します。最下位のバイトはMPUのタイプを示すので、パッチではいきなり68040を示す"4"を返すようにしています。

キャッシュ状態の取得　キャッシュ状態はCACRレジスタを読み出しますが、そのまま値を返すのではなく、下位2ビットのうち、ビット1でデータキャッシュの状態を、ビット0で命令キャッシュの状態を示すように変換し、D0レジスタに返します。

本来は次のようなコードですが、右側のようにパッチしています。

```
            本来のコード                 68040のパッチ
00FFC78C  movec  CACR,d1
00FFC790  ror.l  #1,D1           → swap D1
00FFC792  lsr.w  #7,D1           → rol.w #1,D1
00FFC794  rol.l  #1,D1
```

SRAM設定値にキャッシュモードを設定　これは、SRAMの設定値を読み出した後、キャッシュ設定ルーチンを呼んでいるだけなので、変更はしていません。

キャッシュクリア　キャッシュクリアコードは、先に出たIOCS内部ルーチンで共通に使われているルーチンとは別のルーチンとして用意されています。

これも、CPUSHA IC/DCを使うようにパッチを当てます。

キャッシュモードの設定　キャッシュモードは、D2レジスタの下位2ビットで設定します。ビットの意味は、キャッシュ状態の取得と同様です。

040SYSpatch.xの登録時は、これらに加えて次の機能が追加されます。

本来は次のようなコードでテーブルを引いていますが、右側のようにパッチしています。

```
          本来のルーチン                    68040のパッチ
00FFC7C4    moveq    #$00,D1          → ror.l #,D2
00FFC7C6    add.w    D2,D2            → ror.w #1,D2
00FFC7C8    move.w   $08(PC,D2),D1    → swap D2,D1 / move.l D2,D1
00FFC7CC    movec    D1,CACR
```

040SYSpatch.xの登録時は、これらに加えて次の機能が追加されます。

D1（16進）	機能
8000	040SYSpatch.xの情報の取得
8001	MMUのキャッシュモードの取得
8004	MMUのキャッシュモードの設定
F000	論理アドレスを物理アドレスに変換
F001	指定物理アドレスを指定論理アドレスにマッピング
F002	指定論理アドレスの状態設定／状態取得

040SYSpatch.xの情報の取得　040SYSpatch.xバージョンをD0レジスタに、確保されたROMのコピー領域の先頭番地をA1レジスタに返します。

メモリが12Mバイトの場合には、D0: #"2.1d"、A1: $BF0000となります。

MMUのキャッシュモードの取得　A1レジスタの値が含まれるページ（8Kバイト単位）のキャッシュモードをD0レジスタに返します。D0レジスタに返される値とキャッシュモ

ードの対応は、次のとおりです。

D0	キャッシュモード
0	ライトスルーキャッシュオン
1	コピーバックキャッシュオン
2	キャッシュオフ（シリアライズドアクセス）
3	キャッシュオフ（ノンシリアライズドアクセス）

MMUのキャッシュモード設定　A1レジスタの値が含まれるページ（8Kバイト単位）のキャッシュモードをD2レジスタの値に設定します。D0レジスタには、変更前のモードが返ります。

D2レジスタで設定する値、およびD0レジスタに返される値の意味は、「MMUのキャッシュモードの取得」で示したD0レジスタに返される値と同じです。

なお、MMUでキャッシュオンのモードに設定しても、キャッシュ制御レジスタでキャッシュオフに設定されている場合、すなわち、IOCS-$AC（D1=4）でキャッシュオフにした場合は、キャッシュオフで動作します。

逆に、キャッシュ制御レジスタがオンであっても、MMUでキャッシュオフのモードが設定されている場合、そのページはキャッシュオフでアクセスされます。

論理アドレスを物理アドレスに変換

A1レジスタで示される論理アドレスを物理アドレスに変換したアドレス値をD0レジスタに返します。

指定物理アドレスを指定論理アドレスにマッピング

A1レジスタで示される論理アドレスをD2レジスタで示される物理アドレスにマッピングします。ただし、MMUのページサイズが8Kバイトなので、マッピングで指定できるアドレスは8Kバイト単位で、端数は無視されます。D0レジスタにはマッピングしたページディスクリプタの情報のうち、物理アドレス部を0にした値が返ります。ページディスクリプタの詳細については、68040のユーザーズマニュアルを参照してください。

指定論理アドレスの状態設定／状態取得

A1レジスタで示される論理アドレスが含まれるページの、ページディスクリプタの物理アドレス部を除いた情報を、D2レジスタで示される値に設定します。D2レジスタの内容が－1の場合は状態読み出しとなり、設定は行いません。D0レジスタには変更前のページディスクリプタの情報のうち、物理アドレス部を0にした値が返ります。

■SRAM設定のキャッシュモード設定のパッチ

X68030はキャッシュの設定をSRAMに記憶しておき、このキャッシュモードで起動することができます。

これは、IPLのなかで単にIOCS-$AC（D1=2）を呼び出しているだけなので、IOCS-$ACのパッチが当たっていれば68040起動時も有効になります。しかし、Human68kに対するパッチが当たる前にIPLルーチン内でキャッシュがオンになると、異常動作してしまいます。このため、IPL内の処理をnopでつぶし、かわりに040SYSpatch.xのなかからシステム起動後にIOCS-$AC（D1=2）を呼び出すようにしています。

■Human68k、SCSIドライバ内のパッチ

Human68kおよびSCSIドライバ内にもキャッシュ制御を直接行っている部分があるので、これらのコードを68040対応に直します。

このほかにもキャッシュ制御を自前で行っているプログラムがある場合は、パッチしなければなりません。パッチ方法は、ROMで行ったこととほぼ同様です。

キャッシュ制御コードは、キャッシュ制御レジスタアクセスのためにmovec命令を使いますので、movec命令の機械語コードである$4E7B（16進）をサーチして対応ルーチンをパッチすればよいでしょう。

■コピーバックキャッシュへの配慮

ライトスルーモードのキャッシュは68030と同じなので、機械的にキャッシュ制御コードをパッチするだけですみますが、注意しなければならないのはコピーバックモードがあることです。プログラムコードを書き換えた場合、68030は命令キャッシュをクリアするだけですみますが、68040のコピーバックモードでは書き換えた部分がメモリに書き戻されていない可能性もあります。命令キャッシュのクリアとあわせてデータキャッシュのプッシュを行わなければなりません。

また、キャッシュオフにする場合は、書き戻されずにキャッシュ上にデータが取り残されますので、すぐにキャッシュプッシュして明示的に書き戻さなければなりません。たとえば、Human68k ver3.01でデバイスドライバを呼び出している部分は次のコードになっています。

```
          本来のルーチン             68040のパッチ
 1:      MOVEC   CACR,D0           MOVEC   CACR,D0
 2:      MOVE.L  D0,-(A7)          MOVE.L  D0,-(A7)
 3:      AND.W   #$FEFF,D0         BCLR.L  #31,D0         ←変更
 4:      MOVEC   D0,CACR           MOVEC   D0,CACR
 5:      BSR.W   XXXX              CPUSHA  DC             ←挿入
 6:      MOVE.L  (A7)+,D0          BSR.W   XXXX
 7:      OR.W    #$0800,D0         MOVE.L  (A7)+,D0
 8:      MOVEC   D0,CACR           MOVEC   D0,CACR
 9:      RTS                       RTS
10:XXXX:....                    XXXX:....
```

デバイスドライバを実際にコールしているのは10行目のXXXX：の部分ですが、その前後でキャッシュを操作しています。本来のルーチンは、４行目でCACRのビット８(ED)を０にすることでデータキャッシュをオフにし、デバイスドライバ呼び出しから戻った７行目でCACRの値を戻すとともに、11ビット目（CD）を１にしてデータキャッシュをクリアしています。

しかし、コピーバックキャッシュの場合はそうはいきません。４行目でデータキャッシュをオフにしたら、すぐに５行目でデータキャッシュのプッシュを実行して、キャッシュをメモリに書き戻すようにしています。

5.2.2 MMUの管理

本来、MMUは論理アドレスと物理アドレスの変換をするための機構ですが、040SYSpatch.xではMMUの持つページごとの属性設定という機能を使ってI/O領域のページをキャッシュ禁止にしています。また、ROMのコピー領域を本来のアドレス位置にマッピングしていますが、それ以外の領域は論理アドレスと物理アドレスで一致した値になっています。

メモリエリア	アドレス	キャッシュモード	ユーザーアクセス	書き込み
システムエリア	\$000000～*1	ライトスルー	禁止	許可
フリーエリア	*1～*2	ライトスルー	許可	許可
ROMコピー領域ほか	*2～*3	ライトスルー	禁止	禁止
VRAM	\$C00000～\$E7FFFF	キャッシュ禁止	禁止	許可
I/O	\$E80000～\$EAFFFF	キャッシュ禁止(シリアライズド)	禁止	許可
SPRITE	\$EB0000～\$EBFFFF	キャッシュ禁止	禁止	許可
ユーザーI/O	\$EC0000～\$ECFFFF	キャッシュ禁止(シリアライズド)	禁止	許可
未使用	\$EE0000～\$EFFFFF	キャッシュ禁止	禁止	許可
ROM	\$F00000～\$FFFFFF	ライトスルー	禁止	禁止

＊１：Human68kが使用する領域の直後
＊２：040SYSpatch.xが確保する領域の直前
＊３：メモリの最上位

これらのキャッシュモードは、IOCS-\$AC（D1=\$8004）を使用することで、MMUのページ（８Kバイト）単位で変更することができます。

第6章
終わりに

040turboの製作は、X68000の後継機として登場した機種が、68030という、いわば一世代前のMPUを搭載していたということに我慢がならなくて、X68030でなんとか68040が動くようにしようと個人的に始めたものです。それまでにX68000に68020を載せた経験はありましたが、*1 68040は未知のMPUでした。

そんなわけで、040turboのハードウェアは、試行錯誤でなんとか動くようになったというのが本当のところです。特に、X68030のメインメモリのスタティックカラムモードやVRAMアクセスのタイミングがよくわからず苦労しました。もっとも、X68030のハードウェアタイミングの資料などは公開されていませんから、今でも大丈夫かという不安はあります。動いているからよしとしているというのが本当のところで、厳密にタイミング設計をすること自体、不可能なことです。

実際、最初にバラック基板で組み上げたハードウェアは変更につぐ変更で、今やグチャグチャです。なんとか動くことが確認できたのでプリント基板を起こしましたが、今から見れば、よく動いたな、と驚いてしまうような代物でした。

今の基板は非常に安定して動いていますが、50MHzクロックの取り出しなど、まだ改良の余地があります。また、2次キャッシュやローカルメモリの搭載など、ハードウェアに心得のある人は040turboの基板を叩き台にして、ぜひとも改良を試みてください。

ソフトウェアでも同じようなことがいえます。68040はおろか、68030のキャッシュさえ初体験でしたので、040SYSpatch.sysも試行錯誤で作ってきました。今では040turboの参加者の手により、さらに改良が加えられています。

しかし、今の安定したシステムが最初から提供されていたわけではありません。参加者のみなさんの協力により、ハードウェア、ソフトウェアの両面で改良を続けてきた結果が今のシステムです。

X68030で68040をサポートしていない現在、新しいシステムソフトウェアが出ればパッチは作りなおさなければなりません。また、フリーソフトウェアについても、040turboへの対応をそれぞれの作者に期待するのは無理ですから、自分でパッチを当てていかなければならないでしょう。

040turboのプロジェクトは「フリーハードウェア」と「フリーソフトウェア」から成

＊1
68020の性能がまったく出なかったし、安定もしなかったので、ただ単に動くようになったというだけで、それっきりですが。

り立っております。不具合には可能なかぎり対処するつもりですが、それを保証することはできません。参加者のみなさん自身の責任で使ってください。そのかわり、040turboはソフトもハードもすべて自由です。参加するみなさんの手でよりよいものにしてください。

——P.S.——

68040はそのバス性能に見合った周辺回路でなければ本来の性能を発揮することはできません。このため、040turboによるX68030での68040の使用は、本来のモトローラの設計趣旨からは邪道といえるでしょう。たとえていえば、040turboという強力なエンジンを、ひ弱な足まわりの車体に無理やり載せているようなもので、バランスが悪いためにエンジンの性能を発揮できないのです。

しかし、たとえバランスは悪くても、自分のマシンで68040が動くということにはワクワクするものがあります。そんな思いから040turboのプロジェクトを始めました。いろいろ問題もありましたし、これからも問題が出てくると思いますが、温かい目で見守ってください。

もはや68040さえ速いMPUとはいえなくなってしまいましたが、040turboを搭載したX68030は、間違いなくX680x0シリーズの最高峰であるといえるでしょう。

APPENDIX A
不具合報告のフォーマット

現在わかっている不具合報告は、添付のディスクのなかの「040ERROR.log」というファイルに入っています。ほかにも不具合がありましたら、このフォーマットで報告してください。といっても、あくまで目安です。

以下に記入例を挙げておきます。

040turboの不具合票

発見日　　　　：93/10/15
発見者　　　　：BEEPs (PEG00631)
基板版数　　　：01
68040版数　　 ：02E31F
GAL版数　　　：IC1,IC2,IC3,IC4,IC5
パッチソフト版数：040SYSpatch.sys ver1.1
不具合内容　　：FDアクセスでハングアップすることがある
<その他>
see.xでFDをアクセスすると、ほぼ確実にハングアップする。

なお、この不具合は、040SYSpatch.sys ver2.0で対処しました。

このほか、ハードウェア面では次の不具合が報告されています。

- I/Oスロットを取り付けていないと動かない。
- クロックアップしているマシンでは、ICクリップ接続では動かない。
- 040turboがよくゆるむ。
- 68030の挿し込みが悪くて壊れた。
- 68030モードが動かない（APPENDIX B参照）。
- 純正イーサネットボードが使えない。

APPENDIX B
68030モードの不具合対策

X68030には個体差があるようで、040turboで68030モードを使用すると異状が出る場合があります。X68030のマザーボードに68030を直接挿した場合は問題がなくて、040turbo搭載時にエラーが出る場合はX68030のマザーボードに対策を施す必要があります。

具体的な対処方法は、マザーボード上のIC9の19番ピンと20番ピンの間に小容量のセラミックコンデンサを半田付けします。コンデンサの容量については厳密には調べていませんが、390PFおよび470PFで実績がありますので、この付近の値のものを使用してみてください。

IC9は、内蔵メモリスロットのコネクタの下のほうにあります。ピン間隔の狭い表面実装タイプのICなので、作業は十分慎重に行ってください。

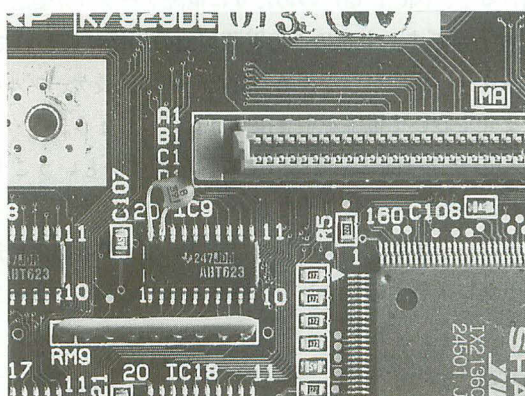



写真　IC9にコンデンサをつけたところ

APPENDIX C
放熱対策

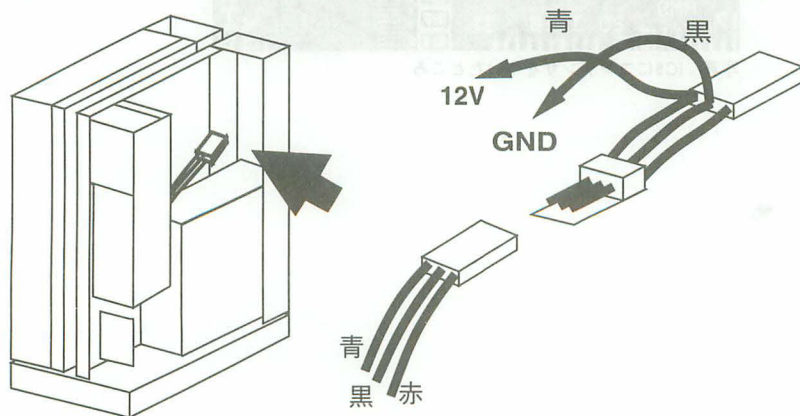
X68030の場合、ヒートシンクによる自然冷却は期待できないので、冷却ファンによる強制冷却を行うほうがよいでしょう。どのように取り付けるかは自由です。

問題は冷却ファンを回す電源をどこからとるかということでしょう。5Vで回るファンなら適当なところからとることもできますが、できればロジックとは別のところからとったほうがいいでしょう。私は、ファン専用に小さな電源を用意しています。X68030のACアウトレットは連動しないので、うっかりファンの電源を入れ忘れて青くなったこともありますが。

もっとスマートにしたい人は、フロッピーや電源が入っている左側のタワーから電線を延ばして電源を持ってくるのがよいでしょう。

左側のタワーを開けると、図C.1 (a) のあたりに電源ケーブルが延びていますので、ここから引き出すのがお手軽です。すでにこの電源が他の用途に使われているなら、図C.1 (b) のように電源コネクタの雄雌をつなぎあわせた一種の延長ケーブルのようなものを作り、これを途中にかませておいて電源を引き出せばよいでしょう。

とはいっても、2つのタワー間にどうやって電線を通すかということのほうがかえって大変でしょう。これは各自で工夫してください。

なお、右タワーにカバーをすると相当に熱がたまるので、68040の付近のカバーに穴を開けたり、タワーの上部に排熱用のファンを設置するなどの対策もとったほうがよいでしょう。

APPENDIX D
040turbo図面

040turbo（01版）の部品表、回路図、実装図、パターン図です。

品名記号	型式	規格	メーカー	単位数	備考
MPU2	XC68040HRC25	25MHz	モトローラ	1	
IC1～3 IC4、5 IC6 IC7～12 IC13、14	GAL16V8B-7LP GAL20V8B-7LP SN74AS74NS SN74AS245NS SN74AS374NS	70nsec 70nsec	ラティス ラティス T.I. T.I. T.I.	3 2 1 6 2	 74F245でも可 74F374でも可
R1、2、4、6～10 R3、5 RA1	BCR 1/10 103J BCR 1/10 201J M9-1-103	10kΩ 200Ω 10kΩ	ベックマン ベックマン ベックマン	8 2 1	
C2—20 C追加	GRM40F104Z50PTB ECEV0JA101SP		村田 松下	19 3	
MPU1用ソケット MPU2用ソケット SOCKET用ソケット IC1～3用ソケット IC4、5用ソケット	IC122-12813-0BS4 IC122-17918-1BS4 PS12813-0B4 IC26-2003-GS4 IC26-2403-GS4	128pin 179pin 128pin、20mm 20pin 24pin	ヤマイチ ヤマイチ ヤマイチ ヤマイチ ヤマイチ	1 1 1 3 2	
CN1～3	IL-2P-S3EN2	JAE	JAE	3	
CN1～3(ケーブル側) CN1～3用コンタクト スイッチ LED	IL-2S-S3L IL-C2-10000 MS-240青 DB-9赤クローム		JAE JAE ミヤマ サトーパーツ	3 10 1 1	付属ケーブル用

表D.1　040turbo部品表

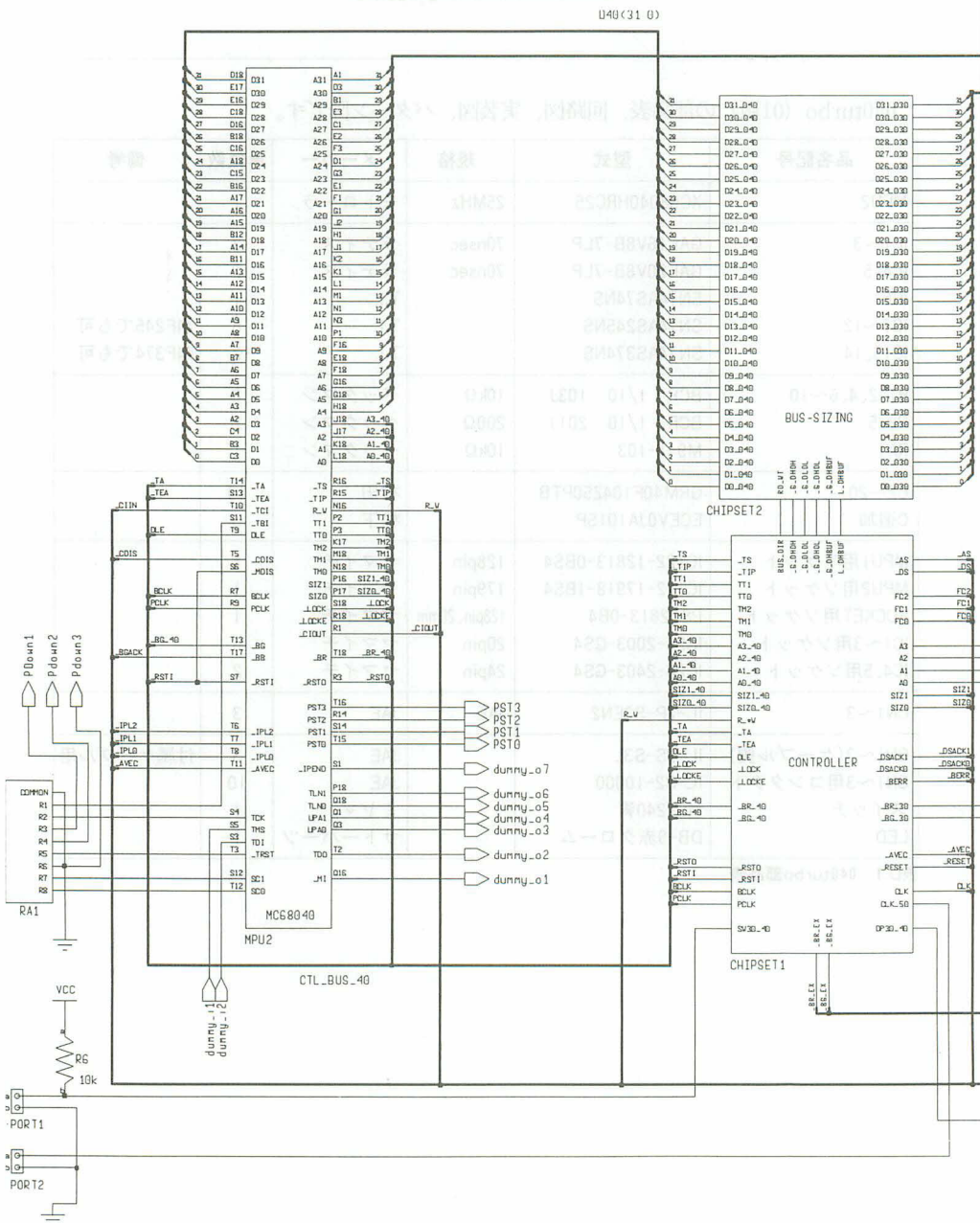



図 1 040turbo全体図（左）

図 2　040turbo全体図（右）

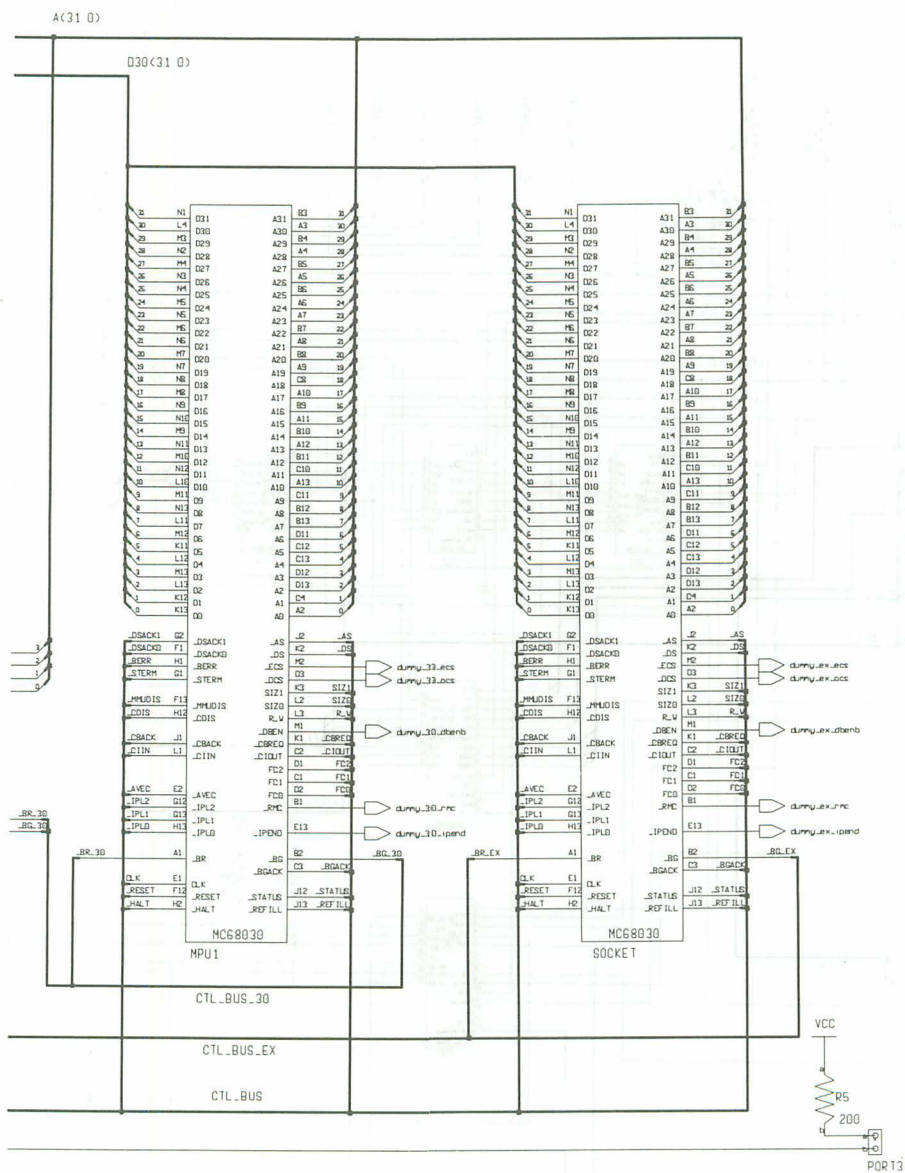
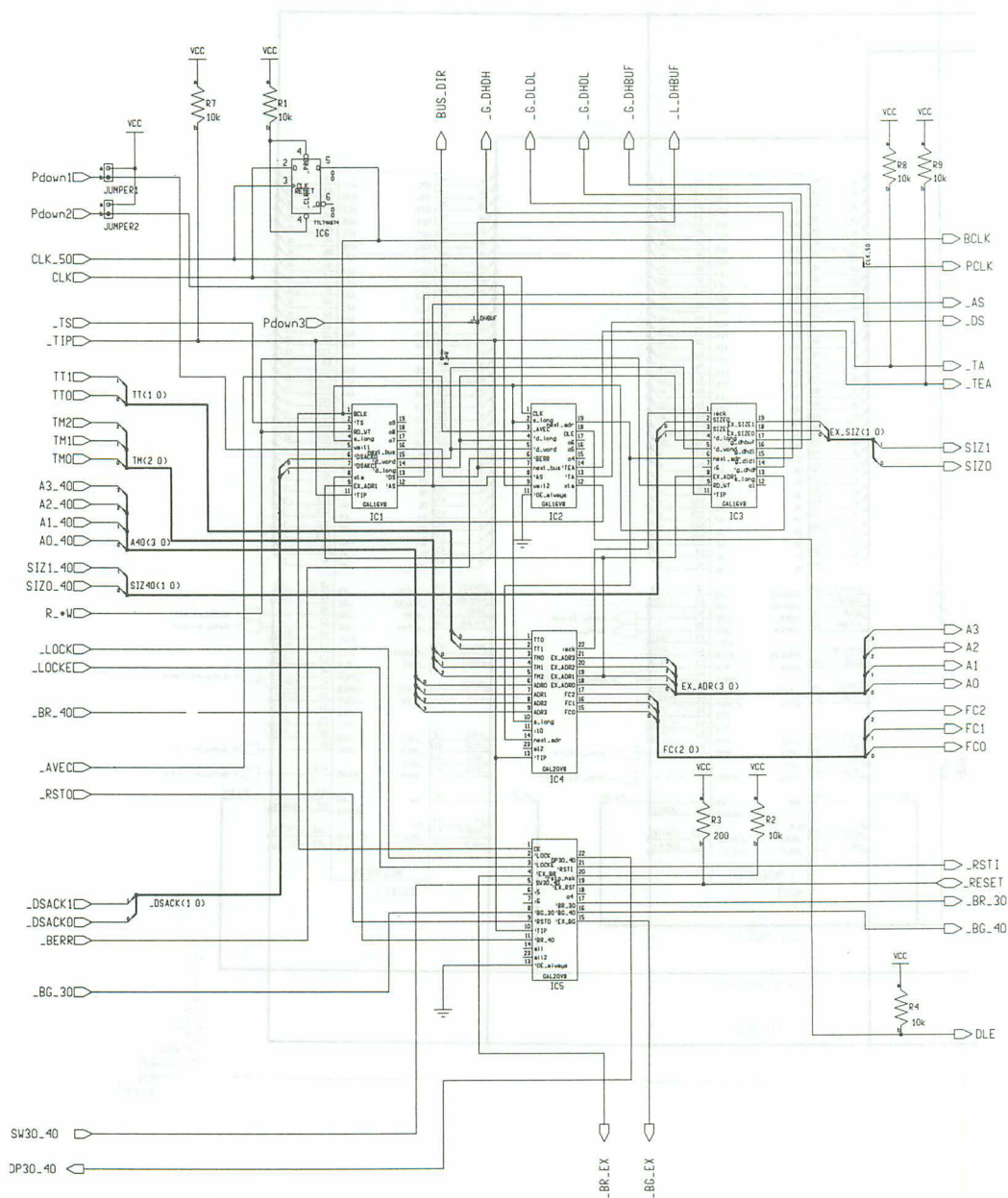
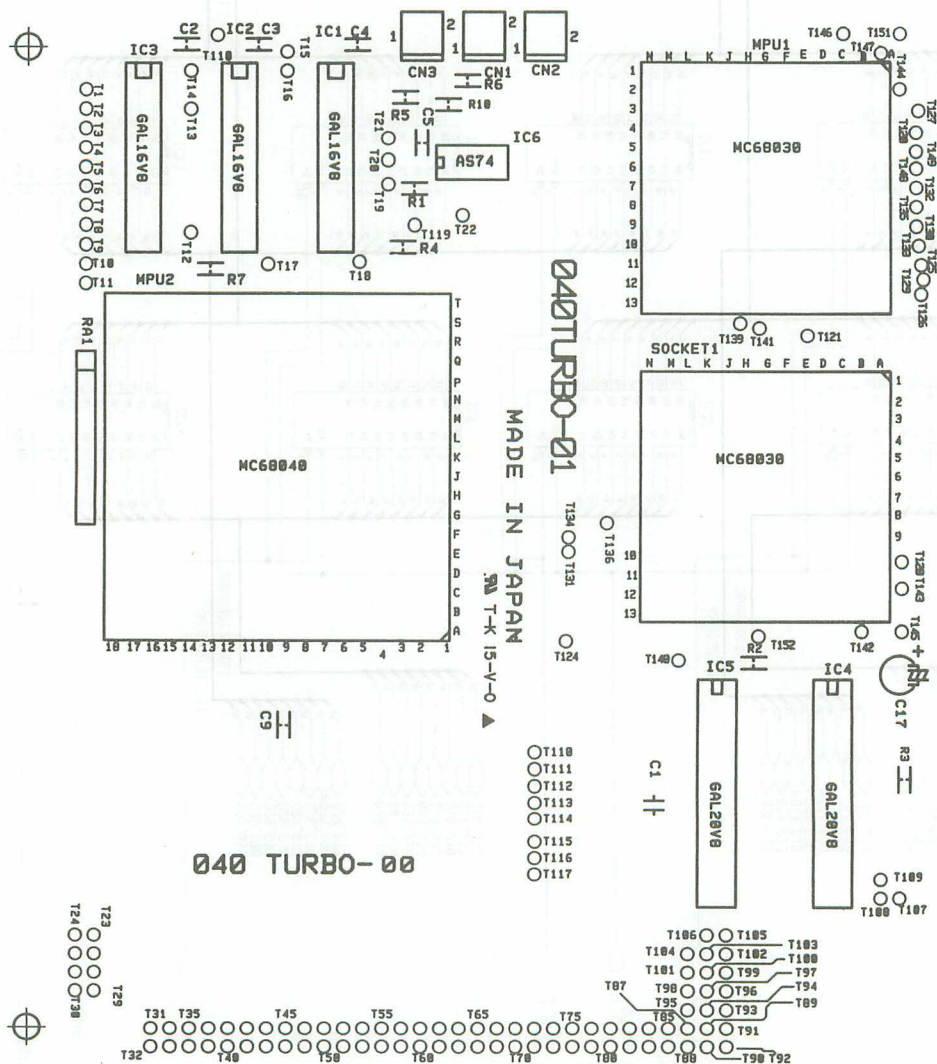
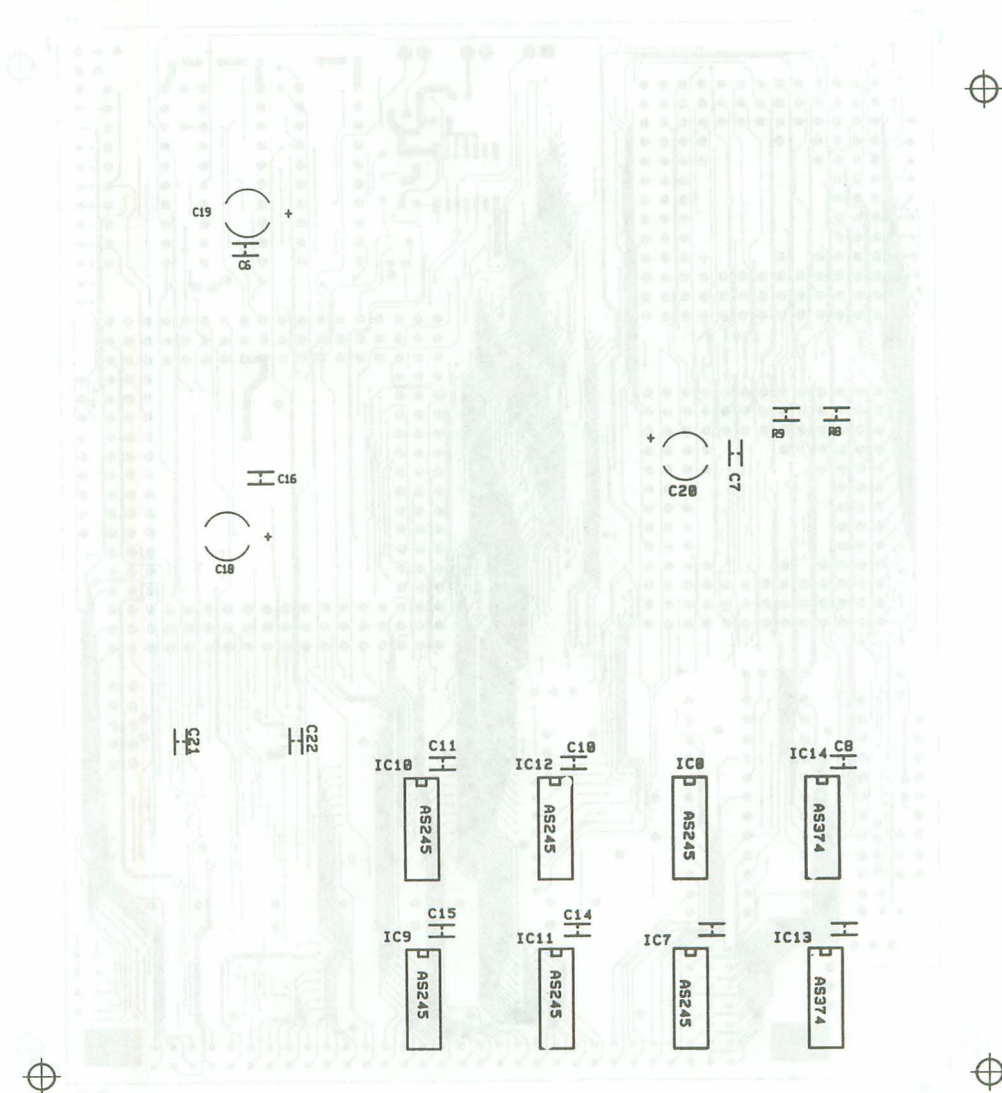



図 3 040turbo制御部（CONTROLLER）

図 4 040turboバス組み換え部（BUS-SIZING）

CF 040TURBO-01-0 DLP
IC3
C2
IC2 C3
IC1 C4
CN3
CN1
CN2
R6
R5
R10
C5
IC6
AS74
R1
R4
MPU1
MC68030
GAL16V8
MPU2
R7
RA1
MC68040
040TURBO-01
MADE IN JAPAN
SOCKET1
MC68030
IC5
IC4
R2
C17
R3
C1
GAL20V8
C9
040 TURBO-00

図D.5 部品面配置

CB 040TURBO-01-0 DLP
C19
C6
C16
C18
C20
C7
R9
R8
C21
C22
C11
IC10
AS245
C10
IC12
AS245
IC8
AS245
IC14
C8
AS374
C15
IC9
AS245
C14
IC11
AS245
IC7
AS245
IC13
AS374

図D.6 半田面配置

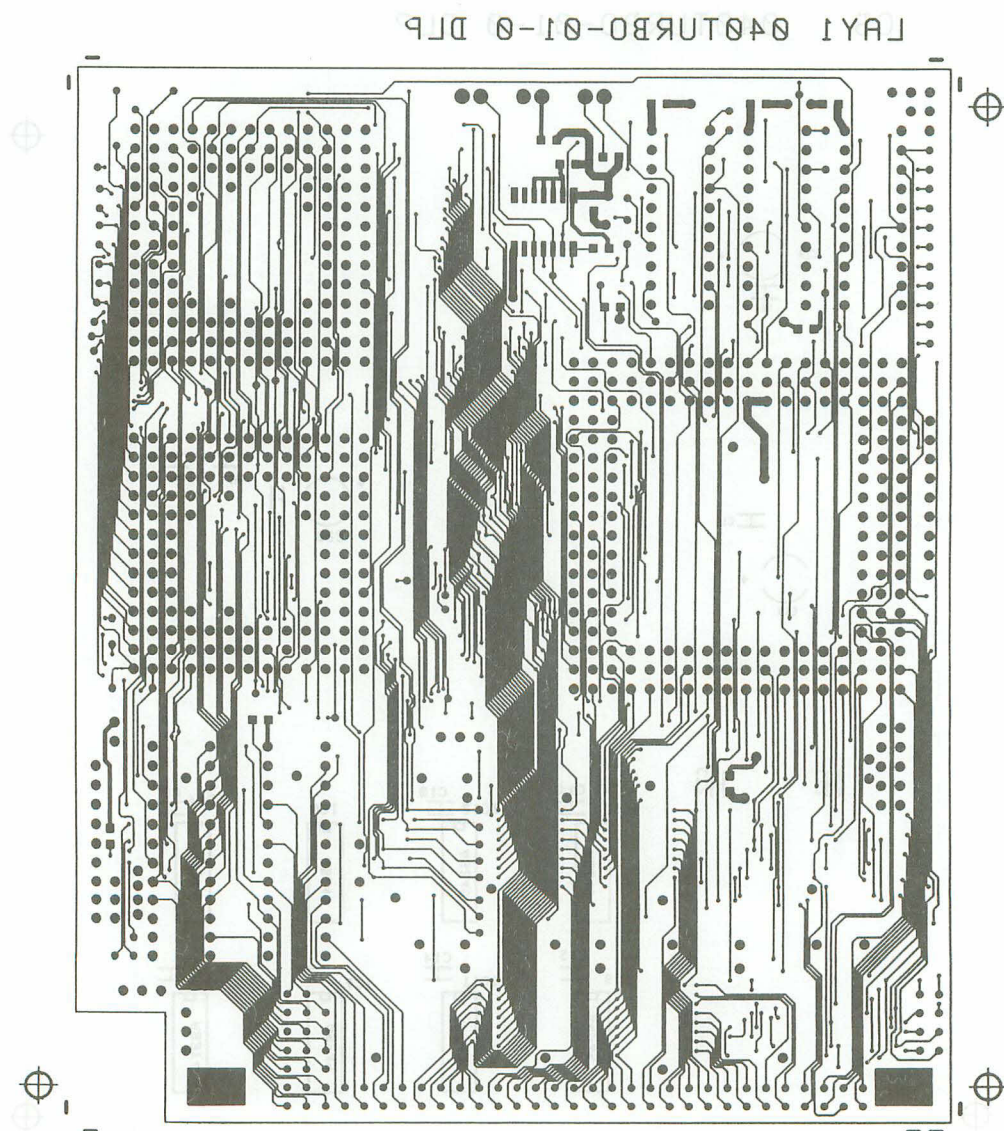



図 7 部品面パターン

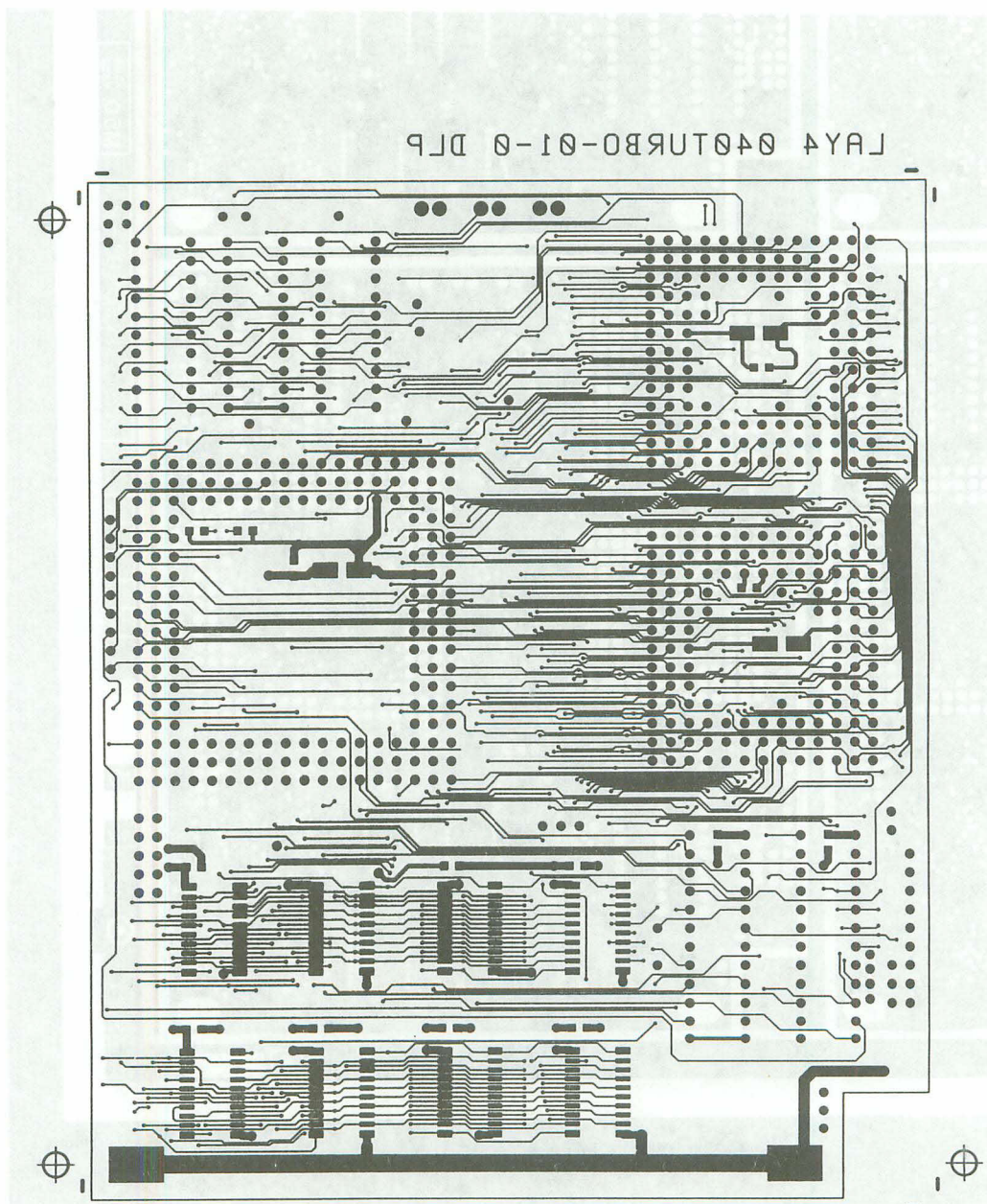



図 8 半田面パターン

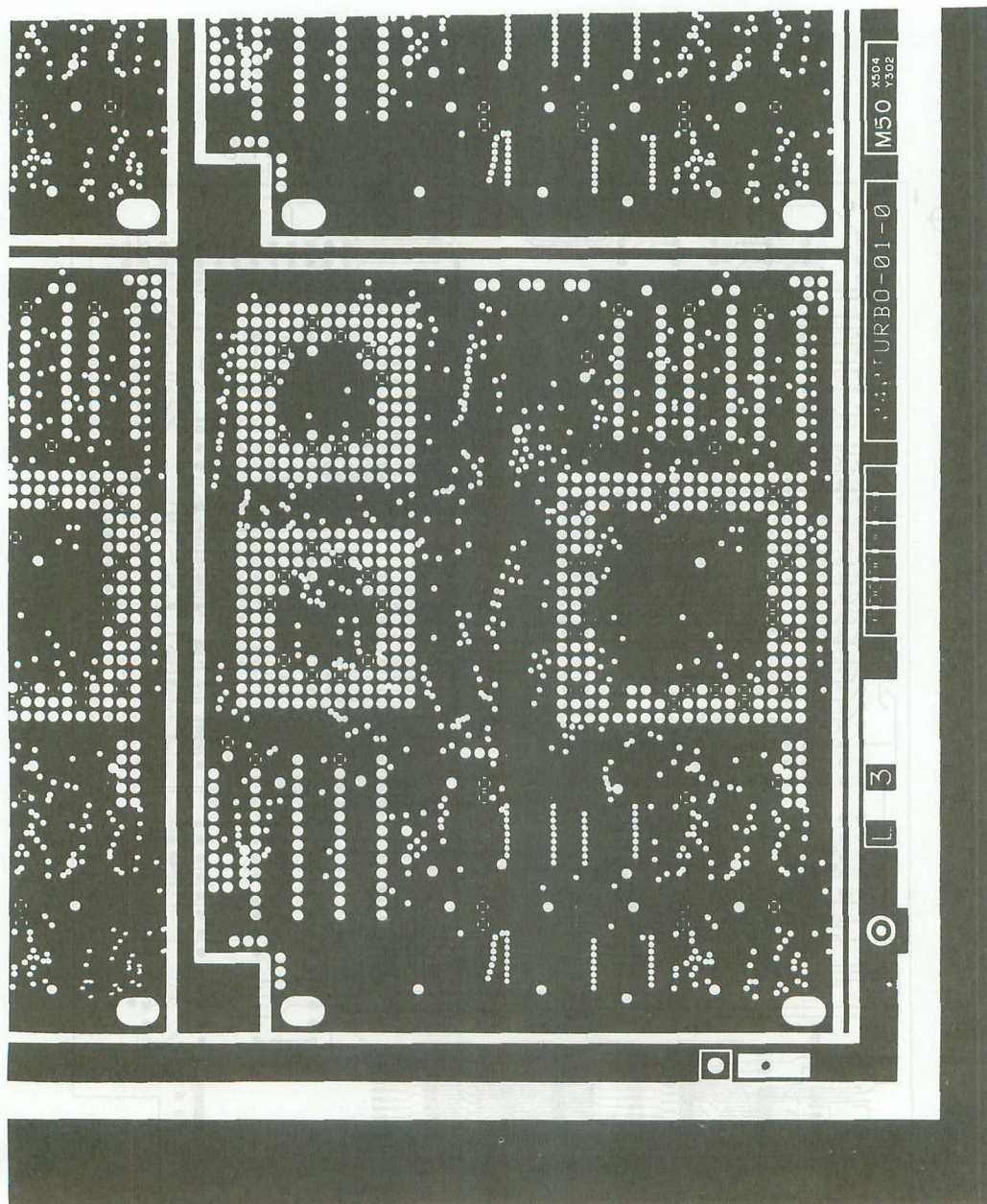



図-9　VCC層パターン

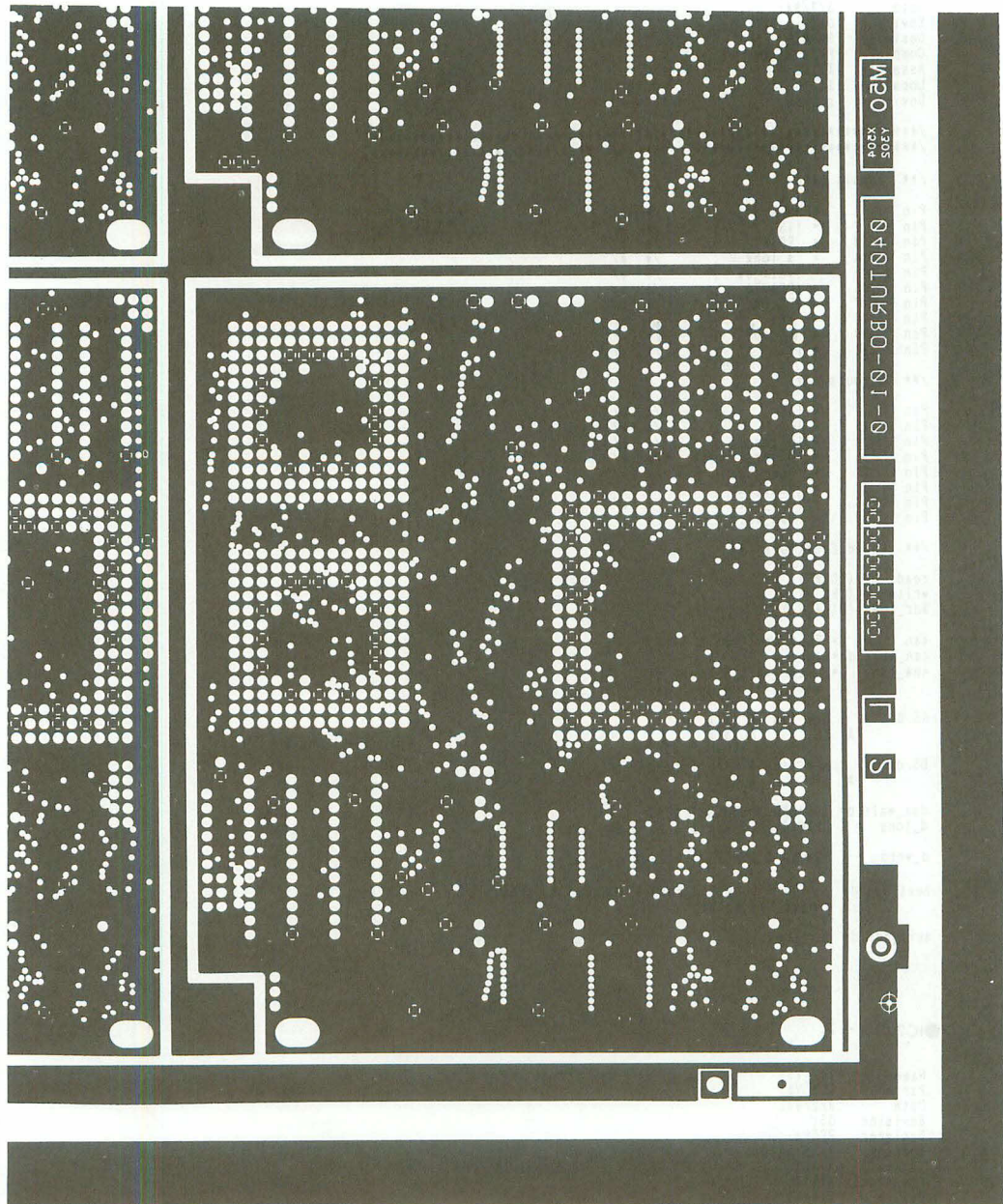



図 10 GND層パターン

040turboのGALのソースリスト

●IC1 のソース

```
Name       IC1_V5;
Partno     IC1_V5;
Date       3/3/94;
Revision   05;
Designer   BEEPs;
Company    FREE HARD;
Assembly   XXXXX;
Location   XXXXX;
Device     g16v8a;

/******************************************************************/
/******************************************************************/

/**  Inputs  **/

Pin     1     = BCLK        ;      /*  */
Pin     2     = !TS         ;      /*  */
Pin     3     =  RD_WT      ;      /*  */
Pin     4     =  s_long     ;      /*  */
Pin     5     = !wait_sw    ;      /*  */
Pin     6     = !DSACK0     ;      /*  */
Pin     7     = !DSACK1     ;      /*  */
Pin     8     =  xta        ;      /*  */
Pin     9     =  EX_ADR1    ;      /*  */
Pin    11     = !TIP        ;      /*  */

/**  Outputs  **/

Pin    12     = !AS         ;      /*  */
Pin    13     = !DS         ;      /*  */
Pin    14     = !d_long     ;      /*  */
Pin    15     = !d_word     ;      /*  */
Pin    16     =  next_bus   ;      /*  */
Pin    17     =  next_adr   ;      /*  */
Pin    18     = !dsa_wait   ;      /*  */
Pin    19     = !tsn_wait   ;      /*  */

/**  Logic Equations  **/

read   = (RD_WT)  ;
write  = (!RD_WT) ;
adr_low=(!EX_ADR1);

tsn         = TS # next_bus & !xta;
tsn_wait.d  = tsn;
new_tsn     =   tsn_wait &  wait_sw
              # tsn      & !wait_sw ;

AS.d    =    new_tsn  & !xta
          #  AS       & !xta ;

DS.d    =    new_tsn  & !xta & read
          #  AS       & !xta ;

dsa_wait.d= (DSACK0 # DSACK1) & AS;
d_long   = ( DSACK0 & DSACK1 ) & AS & dsa_wait;

d_word   = ( DSACK0 $ DSACK1 ) & AS & dsa_wait;

next_bus.d=   AS & s_long & d_word & (DSACK0 $ DSACK1) & adr_low & xta
            # next_bus & !xta ;

next_adr.d= next_bus;
```

●IC2 のソース

```
Name       IC2_V5;
Partno     IC2_V5;
Date       3/2/94;
Revision   05;
Designer   BEEPs;
Company    FREE HARD;
Assembly   XXXXX;
Location   XXXXX;
Device     g16v8a;

/******************************************************************/
/******************************************************************/
```

```
/** Inputs **/

Pin    1    =  _BCLK    :    /*  */
Pin    2    =  s_long   :    /*  */
Pin    3    = !AVEC     :    /*  */
Pin    4    = !d_long   :    /*  */
Pin    5    = !d_word   :    /*  */
Pin    6    = !BERR     :    /*  */
Pin    7    =  next_bus :    /*  */
Pin    8    = !AS       :    /*  */
Pin    9    = !wait_sw  :    /*  */

/** Outputs **/

Pin   11    = !oe_always:    /*  */
Pin   12    =  xta      :    /*  */
Pin   13    = !TA       :    /*  */
Pin   14    = !TEA      :    /*  */
Pin   15    = !as_mask  :    /*  */
Pin   16    =  ta_sub   :    /*  */
Pin   17    = !dsa_wait :    /*  */
Pin   18    =  DLE      :    /*  */
Pin   19    =  next_adr :    /*  */

/** Logic Equations **/

s_word = !s_long;

as_mask.d = AS;
dsa_wait.d = ( d_long # d_word );

wait_mask = !wait_sw # dsa_wait;

xta.d  =  AS & (   (d_long # d_word) & wait_mask
                 # ( BERR  # AVEC )  & as_mask );

ta_sub.d =  AS & (  s_word & d_word
                 #  s_long & d_word & next_bus ) & wait_mask;

TA.d   =    (( d_long & wait_mask ) # ta_sub ) & !AVEC
          # as_mask &  AVEC;


TEA.d  =  AS & BERR & as_mask;
next_adr.d = next_bus;

DLE = AS;
```

●IC3 のソース

```
Name        IC3;
Partno      XXXXX;
Date        10/3/93;
Revision    03;
Designer    BEEPs;
Company     FREE HARD;
Assembly    XXXXX;
Location    XXXXX;
Device      g16v8a;

/*******************************************************************/
/*******************************************************************/

/** Inputs **/

Pin    1    =  iack     :    /*  */
Pin    2    =  SIZE0    :    /*  */
Pin    3    =  SIZE1    :    /*  */
Pin    4    =  !d_long  :    /*  */
Pin    5    =  !d_word  :    /*  */
Pin    6    =  next_adr :    /*  */
Pin    7    =  next_bus :    /*  */
Pin    8    =  EX_ADR1  :    /*  */
Pin    9    =  RD_WT    :    /*  */

/** Outputs **/

Pin   11    = !TIP      :    /*  */
Pin   12    =  ol       :    /*  */
Pin   13    =  s_long   :    /*  */
Pin   14    = !g_dhdh   :    /*  */
Pin   15    = !g_dldl   :    /*  */
Pin   16    = !g_dhdl   :    /*  */
Pin   17    = !g_dhbuf  :    /*  */
Pin   18    =  EX_SIZE0 :    /*  */
Pin   19    =  EX_SIZE1 :    /*  */
```

```
/** Logic Equations **/

read  = (RD_WT)  ;
write  = (!RD_WT) ;
adr_low = (!EX_ADR1);
adr_hi  = (EX_ADR1);
read_adr_hi = read & ( iack # (!iack & adr_hi));

gate  =  TIP & !next_adr;
gate2 =  TIP &  next_adr;

tmp_long    = SIZE0 & SIZE1  # !SIZE0 & !SIZE1;

s_long    = tmp_long;

g_dhdh    = gate & (    s_long
                     #  !s_long & adr_low );

g_dldl    = gate & (    s_long
                     #  !s_long & write & adr_hi
                     #  !s_long & read_adr_hi  & !d_word );

g_dhdl    = gate & (    !s_long & write & adr_hi

                     #  !s_long & read_adr_hi  & d_word )
             # gate2;

g_dhbuf   = gate2 & read;


s0 = SIZE0;
s1 = SIZE1;
s2 = adr_hi;

FIELD size30 = [s2..s0];
FIELD size40 = [EX_SIZE1,EX_SIZE0];

TABLE size30 => size40 {
       'b'000 => 'b'00;  /* long */
       'b'001 => 'b'01;  /* byte */
       'b'010 => 'b'10;  /* word */
       'b'011 => 'b'00;  /* line */
       'b'100 => 'b'10;  /* long->word */
       'b'101 => 'b'01;  /* byte */
       'b'110 => 'b'10;  /* word */
       'b'111 => 'b'10;  /* line->word */
    }

s_long.oe = 'b'1;
g_dhdh.oe = 'b'1;
g_dldl.oe = 'b'1;
g_dhdl.oe = 'b'1;
g_dhbuf.oe = 'b'1;
EX_SIZE0.oe = TIP;
EX_SIZE1.oe = TIP;
```

●IC4 のソース

```
Name       IC4_v3;
Partno     IC4_v3;
Date       11/05/93;
Revision   05;
Designer   BEEPs;
Company    FREE HARD;
Assembly   XXXXX;
Location   XXXXX;
Device     g20v8a;

/*********************************************************************/
/*********************************************************************/

/** Inputs **/

Pin    1   =  TT0      ;       /* */
Pin    2   =  TT1      ;       /* */
Pin    3   =  TM0      ;       /* */
Pin    4   =  TM1      ;       /* */
Pin    5   =  TM2      ;       /* */
Pin    6   =  ADR0     ;       /* */
Pin    7   =  ADR1     ;       /* */
Pin    8   =  ADR2     ;       /* */
Pin    9   =  ADR3     ;       /* */
Pin    10  =  s_long   ;       /* */
Pin    11  =  next_bus;        /* */
```

```
Pin     13    =  !oe       ;      /* */
Pin     14    =  next_adr;        /* */
Pin     23    =  a12       ;      /* */

/**  Outputs  **/

Pin     15    =  FC0       ;      /* */
Pin     16    =  FC1       ;      /* */
Pin     17    =  FC2       ;      /* */
Pin     18    =  EX_ADR0 ;        /* */
Pin     19    =  EX_ADR1 ;        /* */
Pin     20    =  EX_ADR2 ;        /* */
Pin     21    =  EX_ADR3 ;        /* */
Pin     22    =  iack      ;      /* */

/**  Logic Equations  **/

mode_iack = ( TT0 & TT1 );

a3=s_long;
a2=next_adr;
a1=ADR1;
a0=ADR0;

FIELD tm040 = [TM2..0];
FIELD fc030 = [TmFc2..0];
FIELD adr040 = [a3..a0];
FIELD adr030 = [adr1,adr0];

TABLE tm040  => fc030 {
      'b'000 => 'b'101;   /*** V3 ***/
      'b'001 => 'b'001;
      'b'010 => 'b'010;
      'b'011 => 'b'101;   /*** v2 ***/
      'b'100 => 'b'110;   /*** v2 ***/
      'b'101 => 'b'101;
      'b'110 => 'b'110;
      'b'111 => 'b'111;
   }

TABLE adr040 => adr030 {
      'b'0000 => 'b'00; /* not long (!s_long) */
      'b'0001 => 'b'01;
      'b'0010 => 'b'10;
      'b'0011 => 'b'11;
      'b'0100 => 'b'00;
      'b'0101 => 'b'01;
      'b'0110 => 'b'10;
      'b'0111 => 'b'11;
      'b'1000 => 'b'00; /* normal long (s_long & !next_adr) */
      'b'1001 => 'b'00;
      'b'1010 => 'b'00;
      'b'1011 => 'b'00;
      'b'1100 => 'b'10; /* next long (s_long & next_adr) */
      'b'1101 => 'b'10;
      'b'1110 => 'b'10;
      'b'1111 => 'b'10;
   }

iack   = mode_iack;

FC2    = mode_iack # TmFc2;
FC1    = mode_iack # TmFc1;
FC0    = mode_iack # TmFc0;

EX_ADR3 = mode_iack & TM2  # !mode_iack & ADR3;
EX_ADR2 = mode_iack & TM1  # !mode_iack & ADR2;
EX_ADR1 = mode_iack & TM0  # !mode_iack & adr1;
EX_ADR0 = mode_iack & 'b'1 # !mode_iack & adr0;

FC0.oe= oe;
FC1.oe= oe;
FC2.oe= oe;
EX_ADR0.oe= oe;
EX_ADR1.oe= oe;
EX_ADR2.oe= oe;
EX_ADR3.oe = oe;
```

●IC5 のソース

```
Name        IC5;
Partno      XXXXX;
Date        10/3/93;
Revision    03;
Designer    BEEPs;
Company     FREE HARD;
```

```
Assembly   XXXXX;
Location   XXXXX;
Device     g20v8a;

/**********************************************************************/
/**********************************************************************/

/**  Inputs  **/

Pin     1    =  CK        :        /*  */
Pin     2    =  !LOCK     :        /*  */
Pin     3    =  !LOCKE    :        /*  */
Pin     4    =  !EX_BR    :        /*  */
Pin     5    =  SW30_40   :        /*  */
Pin     6    =  i4        :       /*  */
Pin     7    =  i5        :       /*  */
Pin     8    =  !BG_30    :        /*  */
Pin     9    =  !RSTO     :        /*  */
Pin    10    =  !TIP      :        /*  */
Pin    11    =  !BR_40    :        /*  */

Pin    13    =  !oe       :        /*  */
Pin    14    =  all       :        /*  */
Pin    23    =  all2      :        /*  */

/**  Outputs  **/

Pin    15    =  !EX_BG    :        /*  */
Pin    16    =  !BG_40    :        /*  */
Pin    17    =  !BR_30    :        /*  */
Pin    18    =  !bg_wait  :        /*  */
Pin    19    =  !EX_RST   :        /*  */
Pin    20    =  !rsto_msk :        /*  */
Pin    21    =  !RSTI     :        /*  */
Pin    22    =   now_mode :        /*  */

/**  Logic Equations  **/

now_mode.d =    EX_RST & SW30_40
              # !EX_RST & now_mode;

mode_30 = (now_mode);
mode_40 = (!now_mode);

BG_40  =     mode_40 & ( !EX_BR    #  LOCK & !LOCKE );

BR_30  =     mode_30 & EX_BR
           # !mode_30;

bg_wait.d = !BG_40 & !TIP;
EX_BG  =     mode_40 & ( !BG_40 & !TIP & bg_wait )
           #  mode_30 &  BG_30;
EX_RST      = 'b'1;
EX_RST.oe   = RSTO;
rsto_msk    = RSTO # !TIP & rsto_msk & !EX_BG;

RSTI        = EX_RST & ( !RSTO & !rsto_msk );

BG_40.oe='b'1;
BR_30.oe='b'1;
EX_BG.oe='b'1;
RSTI.oe='b'1;
```

APPENDIX E
クロックアップマシンでのウェイト挿入

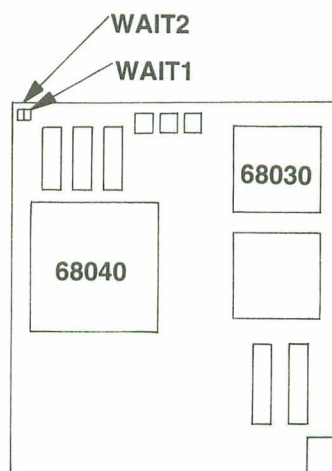
040turboは、25MHzのクロックで動作するようにチューンナップされています。このため、クロックアップしたマシンで使用する場合は正常に動作しません。この問題に対応するため、独自にウェイトを挿入できる機能を追加しています。

以下の設定をすると、ウェイトを挿入することができます。

1）IC1-5番ピンをプルダウン　--->$\overline{\mathrm{TS}}$信号から$\overline{\mathrm{AS}}$信号を作るときに１ウェイト

2）IC2-9番ピンをプルダウン　--->$\overline{\mathrm{DSACKx}}$信号から$\overline{\mathrm{TA}}$信号を作るときに１ウェイト

通常は２）のIC2-9番のほうの１ウェイトだけで十分ですが、それでも調子が悪いようなら、１）のほうも試してみてください。

最新基板（'94年４月以降製造の物）に関しては，図E.1のようなジャンパー端子を設けてあり、ショートプラグを抜けばプルダウンされるようになっています。

図E.1　ジャンパー端子の位置

	WAIT2:IC2-9 DSACKx ► TA	WAIT1:IC1-5 TS ► AS
	VCC 0-wait	VCC 0-wait
	VCC 0-wait	Pull-Down 1-wait
	Pull-down 1-wait	VCC 0-wait
	Pull-down 1-wait	Pull-down 1-wait

図E.2　ショートプラグとウェイトの関係

付録 2
040turboアプリケーションソフトウェア動作状況

('94年 4 月24日現在)

040turboでソフトウェアの互換性に関して問題なるのは、主にキャッシュの違いによるものです。X68000時代のプログラムの一部が、X68030の登場時に不具合を起こしたのと同様、ある程度しかたがありません。しかし、多くのソフトウェアが040turbo上で問題なく動いていますし、不具合のあるものもパッチや対処方法が考えられています。

どの程度のアプリケーションソフトウェアが040turboで動くのかの目安として、040turboの参加者のみなさんに協力していただき、動作しているアプリケーションソフトウェアをリストアップしてみました。

なお、X68000およびX68030のすべてのソフトウェアを網羅しているわけではありませんし、また、あらゆる状況を想定してチェックしているわけでもありません。あくまで、参考と思ってください。

■コピーバックモード、ライトスルーモードともに問題なく動くもの

16color graphic Loader/Saver ver1.07
ask68k.X
bdif/bup ver1.1　rel7/rel12
bmp.x Ver 0.1-00-/A
BMPL.R v0.32
CHGENV Version 0.12
cawf-4.0+lx
CVS v1.3 for Human68k
DCA.R ver 1.28
DCACHE2.R v2.11
de.x ver0.22
DEDIT version 2.25
dis ver 2.06β
disasm X6_04
DJ.x Ver 1.0a X68k Ver 4.2.0
EasyDraw.x
ED.R ver 0.99XH
expfd v0.3
F.X (size 20058　88-09-01　12：00：00 version)
FES.X Ver 1.15e
fish v0.8.1

FIXER Ver4.0-V1.20 Rel.10
flex version 2.3 for Human68k X6_10
g++ version 1.40.3 X68k
gcc version 1.17 Tool #1(X68) Based on 1.42
Ghostscript 2.4000000953674.1
gif.x Ver 2.00
GNU Bison version 1.19 for Human68k X6_01
GNU Make version 3.62 (X6_12)
GNU tar version 1.11.2 (X6_01)
gzip 1.2.4(X6_03 for X68030)
hapic.r V1.29D
HAS.x v3.02
HAR.x v1.33
HIOCS version 1.10+16
Hi-Speed FontDriver v1.20 (hfont/h12fon)
HLK.x v2.28
ITA Toolbox
jlatex (virtex"This is pTeX, C Version 2.99 j1.6 p1.0.9a")
jpeged.r Ver 1.00+0.15
jperl verison 4.036 + 1.4 + 0.0(Human68k, SJIS)
kran.x ver1.3
less version 177 for Human68k
lhes v0.76
lhv v0.84, 0.85'
lhx v.1.04f
lndrv version 1.25
MACS || MIRO Animation Controller for System registration Version 1.04
MADRV verison 2.01a HSX/2.06+15/16
mfged V5.1o
MicroEmacs 3.10 J1.31_N1, J1.43 (rel.5)
mint v1.31
Mirage System
MORPH!
mount87
Mule Version 1.0 (KIRITSUBO) PL01 (X07)
My.x Ver 1.11
NIFP X6_05a (based on Ver.3.71)
Nemacs.x
panic.x version 1.24
patch.x
PD KSH v4.9 (X6_23)
PIC2.x ver0.06
pi.r ver1.07+01
plain2 r2.3, r2.53
preview.x (2p09)
RC play driver v3.00f & converters
RCS version 5.6.4 (beta) X6_8
RJJ Release 0.21
see v0.31
SETPATH Version 1.00+
shake v2.06

SiV.x ver0.97a
sprndev.x
sprndiv2.sys
SV.x Version 0.99 rel.6
swap ver1.20
tf.x 1.58i
tif.x Ver 0.22
TIFFVIEW.X Ver 1.06
tsort.r ver2.55
TwentyOne v1.31
UnZip 5.0p1
WINDEX v1.14
X680x0 TeX Previewer Ver 2p09a
X68k Font Manager Ver 3.00a
X Basic
xgif.x Ver 0.5
XMISA.x Ver 2.13
Zip 2.0
ZMUSIC System V2.01
zsh v2.3.1X6_03p 6
シャーペン
キャンバス

■コピーバックモードでは問題が出るが、ライトスルーモードでは問題なく動くもの

ソフトウェア名	不具合の内容
飢狼伝説 2	コピーバックでも動作するようだが、途中でハングしたり、不安定な動作をしたりする。PCMもおかしくなる。問題なく動くという報告もある。
三國史 2	コピーバックでも動作するようだが、表示がおかしくなり、不安定になる。
adjust2a.r	コピーバックでも動作するようである。
apicg.r ver2.01	ほとんど正常に動く。ただし、256MAGや256BMPなど、768×512ドット　256色表示を無理やり実現するモードでは正常に表示されない。040SYSpatch.sys v2.3以前ではハングしていたが、v2.3以降になってハングしなくなった。
ED.R	コピーバックでも動作するが、起動時にエラーが出ることがある。エラーが出ても「中止」を選んでやれば、とりあえず動くようである。
FMT.R ver 1.42	ドキュメント中でlzxで圧縮することを勧めているが、lzxで圧縮してあるとコピーバックでは動かない。
Geograph Seal	040モードだと画面まわりが少し乱れる。また、クロックを上げると飛ぶ。
Matier V2.00	起動はライトスルーでないと駄目。起動後は、コピーバックに切り替えて動作させられるが、いくつかの機能が正常に動作しないようである。
PCM8.X 0.48b	コピーバックでは動作が不安定になる。

QuTERM	コピーバックでは文字落ちの可能性がある。 これはtmsio.xの不具合に起因するものである。
RCD	クロックを上げると飛ぶ。
RyDeeN　雷電 version 0.04b	PCM8を外せば、コピーバックでも問題なし。
StreetFighter 2	コピーバックでも動作するようだが、途中でハングしたり、不安定な動作をしたりする。PCMもおかしくなる。 なお、キャッシュオフでも駄目という報告もあり。
TMN.x (tmsio.x ver0.31)	コピーバックでは文字落ちの可能性がある。 これはtmsio.xの不具合に起因するものである。
ZsStaff PRO68k version 3	コピーバックのデータキャッシュイネーブルで動かすと、マウスがいうことをきかなくなる。動作はするが、独自のルーチンを使っているようで、マウスボタンON・OFFのルーチンまで高速化してしまい、痙攣技を使わないとダブルクリックを認識してくれない。

■キャッシュオフで動くもの

ソフトウェア名	不具合の内容
グラディウス	キャッシュディセーブルで動く。
DiskCopyII(dc.r) ver0.65	キャッシュオンだとコピーに失敗する。
LED.R	ライトスルーでは画面表示が一部おかしくなるだけで、エディタ自身にはまったく影響がない。コピーバックでも動作することがあるが、確実に動くとはいえない。

■040turboでは個別のパッチが必要なもの

ソフトウェア名	補　足
CDDEV.SYS	patexec用のパッチファイルCDDEV.patが公開されている。
condrv.sys1.09c	X68030用にパッチを当てれば動く。
db.x ver 3	バイナリ差分ファイルdb.bfdが公開されている。
fsx.x	patexec用のパッチファイルfsx.patが公開されている。
ROMデバッガ	SRAMによる起動時の組み込みはできないが、一度040SYSpatchを登録後は使用可能となる。
scd.x ver 3	バイナリ差分ファイルscd.bfdが公開されている。
view.x ver1.15c	X68030用にパッチを当てればOK。

■040turboで不具合のあるもの

ソフトウェア名	補　足
jpeg.x Ver 1.30	動作はするが、一部表示されないデータがある。
mag.r v1.10	ほとんど正常に動く。ただし、256MAGや256BMPなど、768×512ドット　256色表示を無理やり実現するモードでは正常に表示されない。040SYSpatch.sys v2.3以前ではハングしていたが、v2.3以降になってハングしなくなった。
sml（Small MAG loader）ver0.04	ほとんど正常に動く。ただし、256MAGや256BMPなど、768×512ドット　256色表示を無理やり実現するモードでは正常に表示されない。040SYSpatch.sys v2.3以前ではハングしていたが、v2.3以降になってハングしなくなった。
Straight Filer ver.4.10	1. &push_cache &pop_cacheが正常に動かない。 2. ファイル選択画面でスクロール時にアドレスエラーが出る。 それ以外は問題なし。

■その他

ソフトウェア名	補　足
Human68k ver2.5 Human68k ver3.0	パッチプログラム040SYSpatchが対応していないので、キャッシュオンにはできないが、使用可能である。
XOS9	起動はできるが、キャッシュオンにできるかどうかは不明である。

■あとがき

やっぱり68040がいい。

この、最初の思い入れから始まった040turbo計画を、今、1つの形にできたことを誇りに思っています。X68030がわが家にやってきてからほぼ1年、まっとうなパソコンとしての使い方をした記憶はほとんどないのですが、充実した時間を過ごすことができました。巷の、マルチメディアだ、高性能だといった流れとはまったく別の時間を過ごしてきましたが、こういうこともまた、X68030ならではのことなのかもしれません。

最後に、040turbo計画の参加者ならびに応援してくれた方々、私のわがままにつきあって忙しいなか、この本のために寄稿してくれた方々、度重なる設計変更につきあってくれた神峯電子の方々、遅れる原稿に辛抱強くつきあってくれたソフトバンクの方々、そして、陰ながら支えてくれたカミさんと、いつも笑顔の息子*1に感謝して、この本を締めくくります。

＊1
私事ですが、この春、2人目が生まれました。

1994年4月26日　午後7時26分

BEEPs

NIFTY-Serve：PEG00631

MAX-BBS：　MAX0006

●ソフトバンクからのご案内

「X68/040turbo」出版にあたり、読者の方の中で040turboを購入したい方には15名様に限り、040turboを配布させていただきます。配布価格は84,000円です。希望者の方は、読者カードの040turboの配布を「希望する」という欄に○をお書き込みください。6月末日を締切とし、この日までに募集人員を超えた場合は、抽選とさせていただきます。入金方法等については、当選者の方に追ってご連絡致します。

なお、電話でのお問い合わせは受け付けておりませんので、よろしくお願い申し上げます。

■INDEX

数字・記号

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

あ行

か行

さ行

た行

な行

は行

ま行

ら行

わ行

X68/040turbo

1994年5月30日　初版第1刷発行

著　者………BEEPs（ビープス）

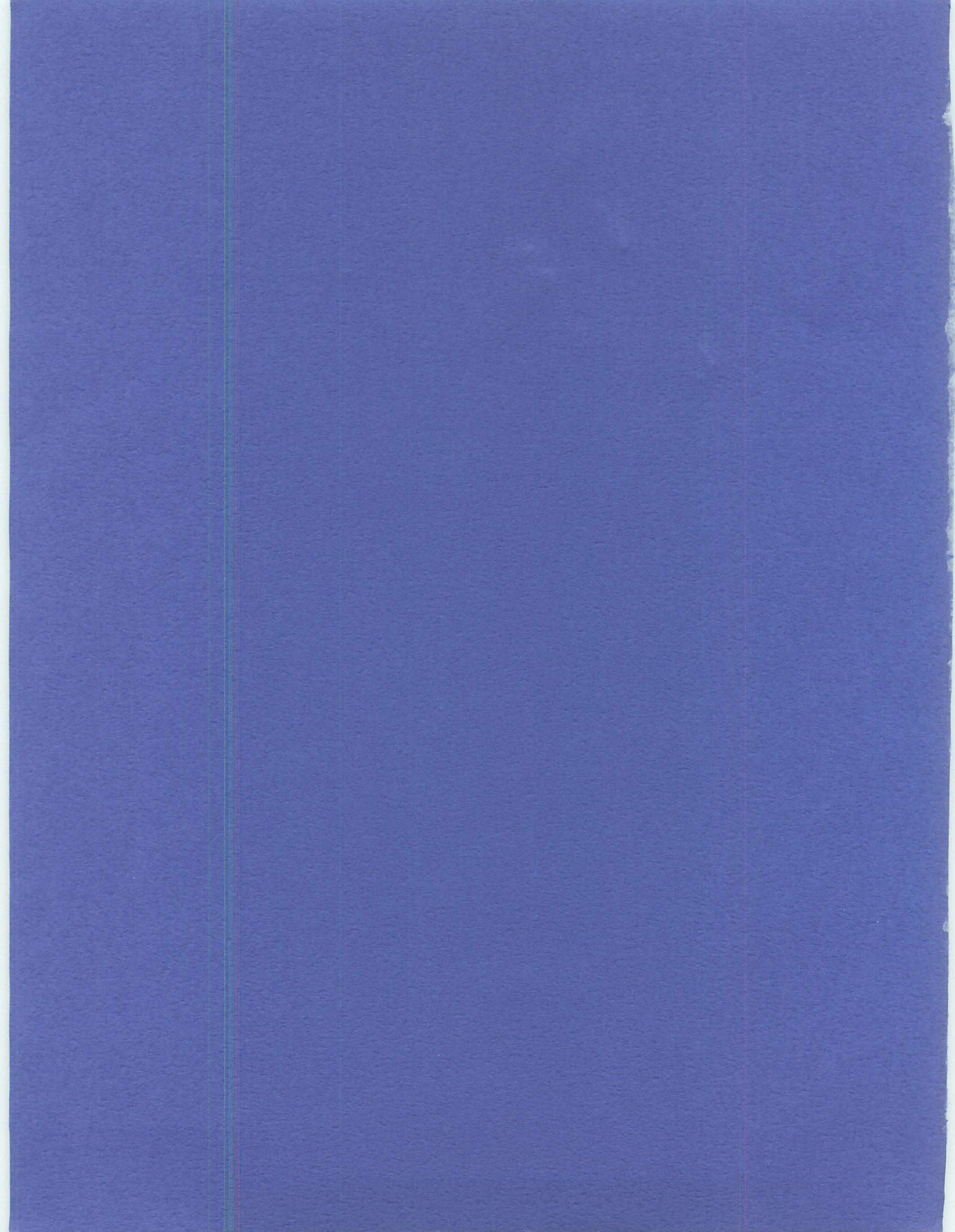
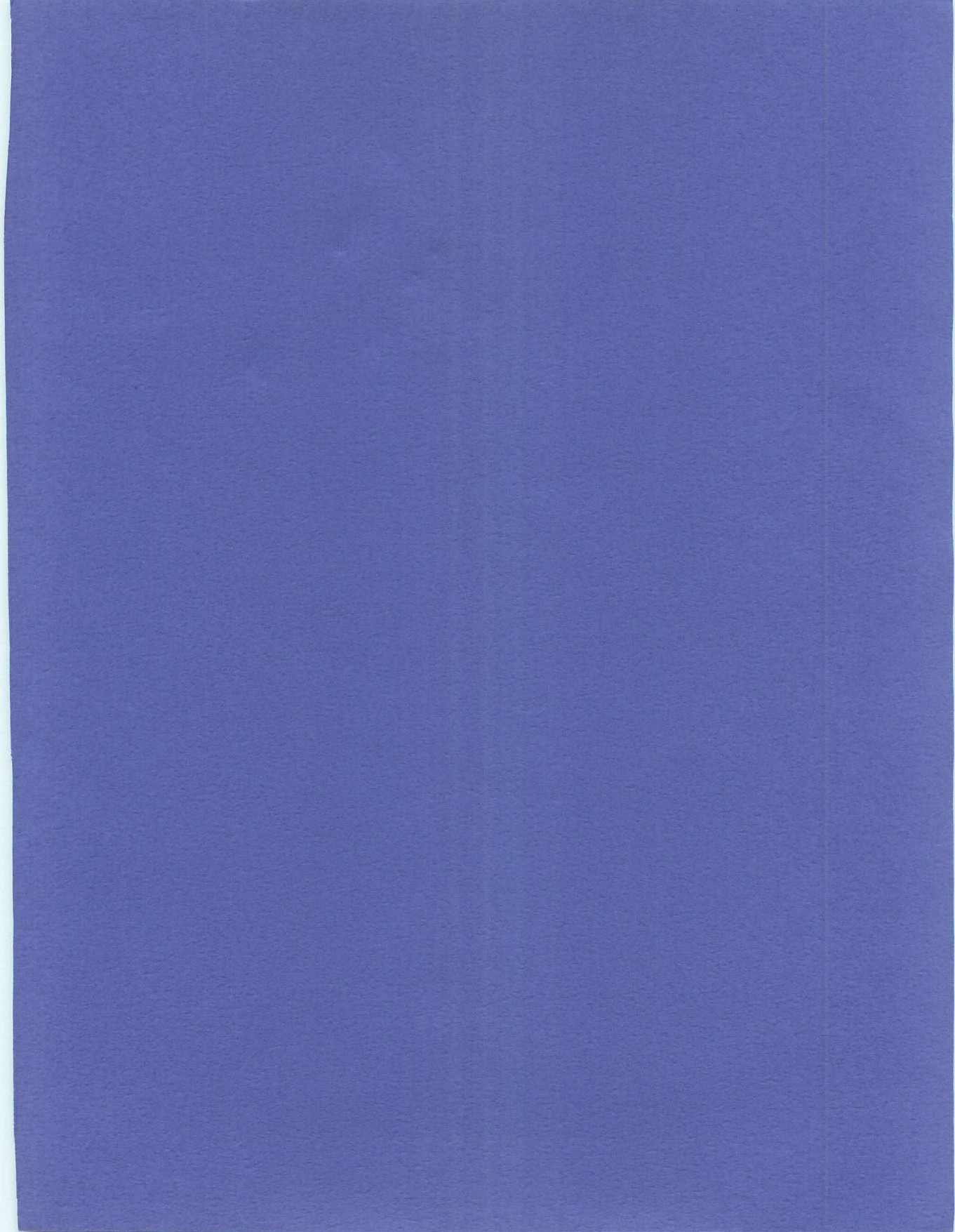
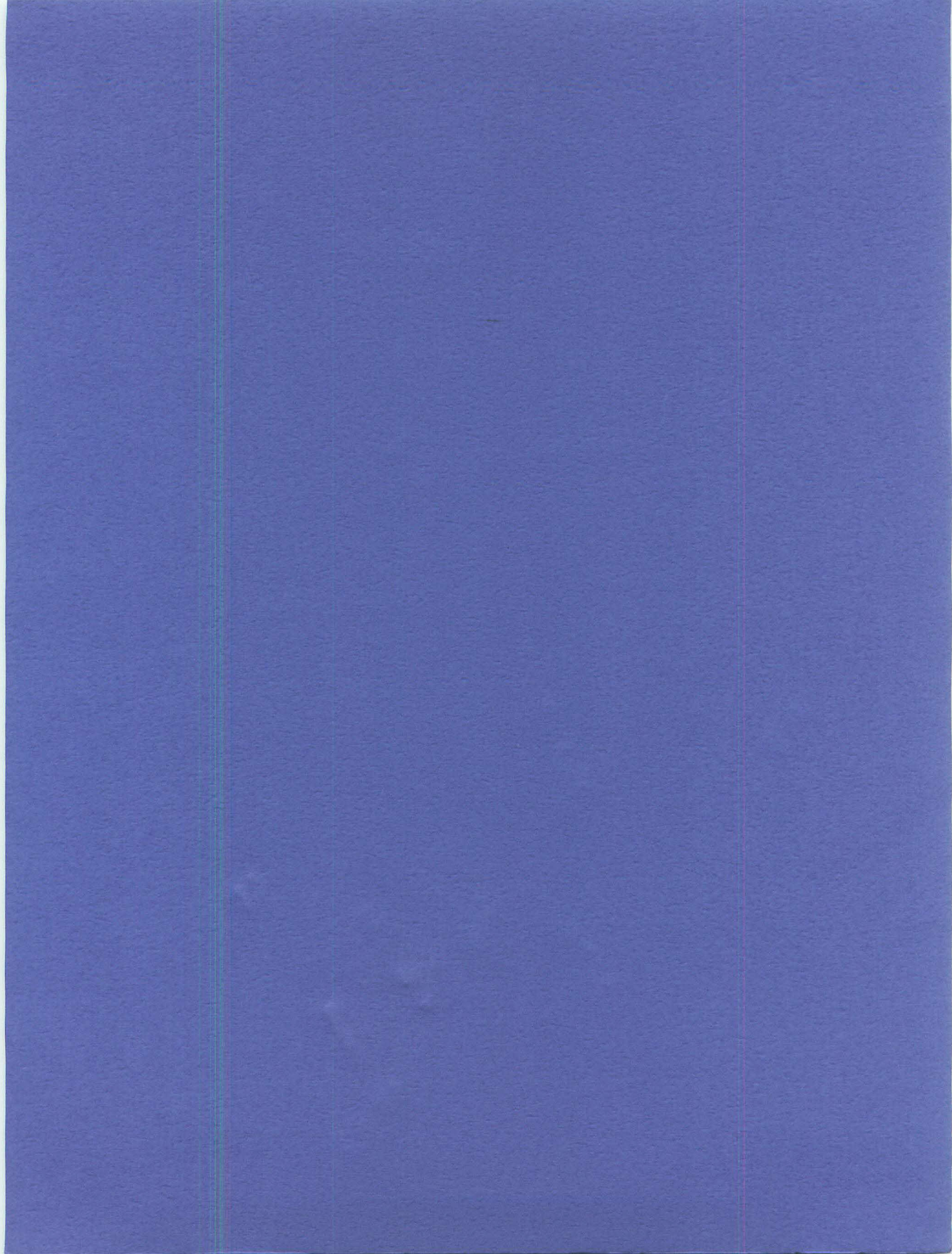
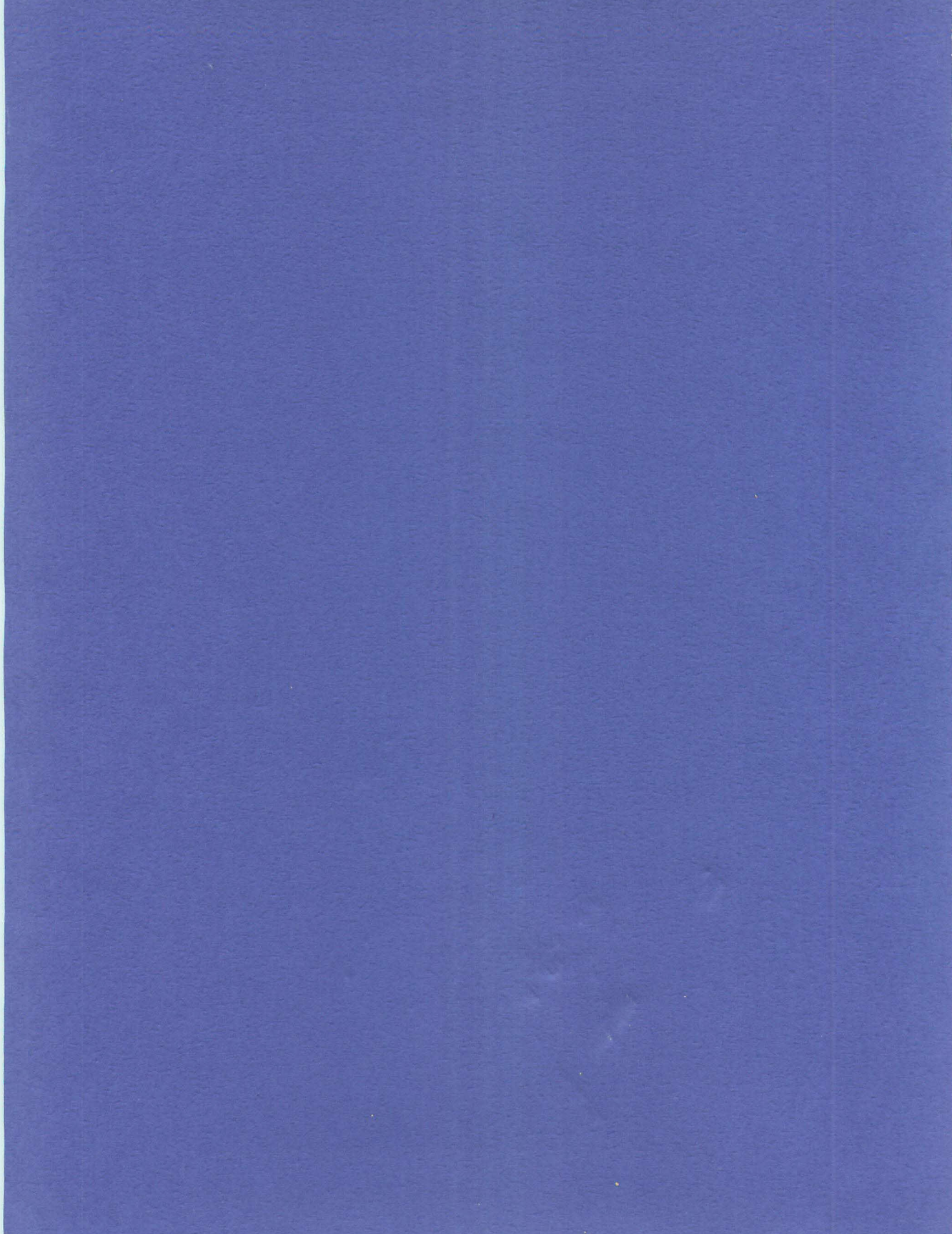
発行者………橋本　五郎

発行所………ソフトバンク株式会社　出版事業部
〒103　東京都中央区日本橋浜町3-42-3
販売　03(5642)8101
編集　03(5642)8140

組　版………帆風

印刷所………株式会社厚徳社

Printed in Japan 1994

ISBN 4-89052-505-X C 0055

郵 便 は が き

料金受取人払

日本橋局
承認

1277

差出有効期間
平成 8 年 4 月
30日まで

1 0 3-0 0

1 6 1

東京都中央区
日本橋浜町3-42-3

ソフトバンク㈱出版事業部

ハードウェア活用書編集部行

住所				
氏名		年齢	性別	男 女
職業・勤務先 学校(学部・学年)		所有機種		

X68/040turbo

弊社ソフトバンクの本をお買上げいただきありがとうございます。今後の編集の資料にさせていただきますので、下記のアンケートにお答え下さい。ご協力をお願いいたします。

■この本を何でお知りになりましたか？

1．雑誌広告（雑誌名／　　　　　　　　　　）
2．雑誌の紹介記事で（雑誌名／　　　　　　　　　　）
3．書店で見て　　4．書店ですすめられて　　5．人にすすめられて
6．その他（　　　　　　　　　）

■この本をお買上げの書店名は？

都・道
府・県　　　　　　　　区・市　　　　　　　　書店

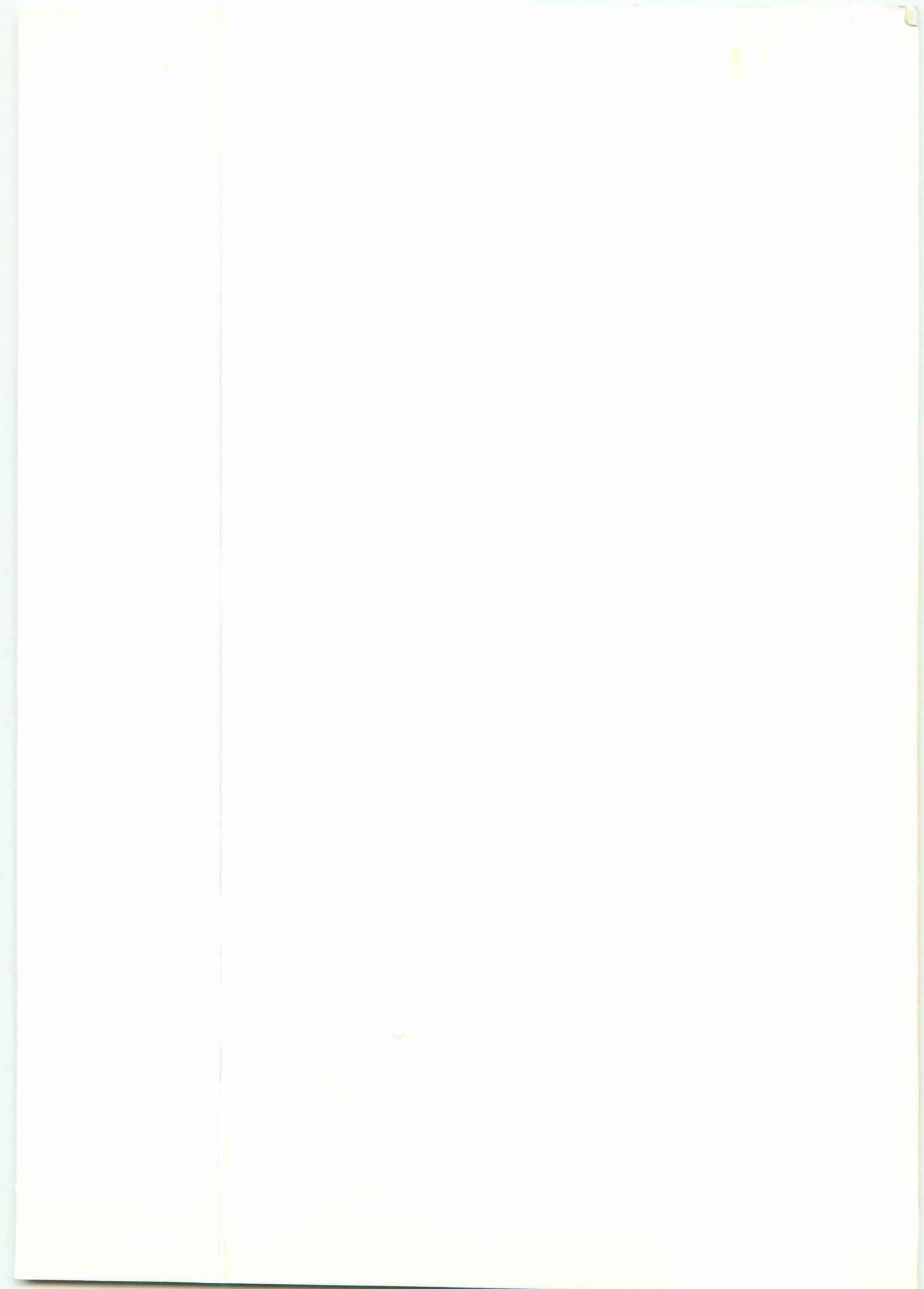
■以下の質問にお答え下さい

内　容	1．おもしろい	2．ふつう	3．つまらない
装　丁	1．よい	2．ふつう	3．わるい
価　格	1．高い	2．ふつう	3．安い
NetBSD本	1．出たら買う	2．買わない	3．わからない
基板配布	1．希望する	2．希望しない	

■お読みになった感想をお聞かせ下さい

■弊社発行の書籍・雑誌をお読みになったことがありますか？

書籍名（　　　　　　　　　）　　雑誌名（　　　　　　　　　）

■今後どのような企画をお望みですか？

ISBN4-89052-505-X

C0055 P2400E

定価2,400円
(本体2,330円)

1910055024007

心からX68kを愛する著者が、X68030購入を機に、これにMPU68040を載せ、

高速・ハイパワーなマシンに変貌させるためのボードを自作することを思い立ちました。

本書は、040turboと名付けられたこの計画を、さまざまな試行錯誤の末に

実現させた男の製作奮闘記です。040turbo取扱説明書付き。